# PARALLEL FUNCTION PROCESSOR

# PROGRAMMER'S MANUAL

SPECIAL TECHNICAL REPORT

REPORT NO. STR-0142-90-006.1

May 14, 1990

---

## GUIDANCE, NAVIGATION AND CONTROL

## DIGITAL EMULATION TECHNOLOGY LABORATORY

Contract No. DASG60-89-C-0142

Sponsored By

The United States Army Strategic Defense Command

---

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology
Atlanta, Georgia 30332 - 0540



---

Contract Data Requirements List Item A004
Period Covered: Not Applicable
Type Report: As Required

## DISCLAIMER

DISCLAIMER STATEMENT - The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

## DISTRIBUTION CONTROL

(1) DISTRIBUTION STATEMENT - Approved for public release; distribution is unlimited.

(2) This material may be reproduced by or for the U.S. Government pursuant to the copyright license under the clause at DFARS 252.227 - 7013, October 1988.

# PARALLEL FUNCTION PROCESSOR

# PROGRAMMER'S MANUAL

May 14, 1990

---


Author

Richard M. Pitts

COMPUTER ENGINEERING RESEARCH LABORATORY

Georgia Institute of Technology
Atlanta, Georgia 30332-0540


---

| | |
|---|---|
| Eugene L. Sanders | Cecil O. Alford |
| USASDC | Georgia Tech |
| Contract Monitor | Project Director |

---

# Preface

This document is Revision 1 to an earlier document report No. STR-0142-90-006. Since the first document was submitted, there have been changes in the software environment to speed up the learning curve. In order to simplify the process of compiling, binding, building, and executing programs several of the commands have been changed. Section 3.3 describes the steps required to complete the process of compiling, binding, building, and executing programs. Appendix G describes these new programming tools. Another change was to combine the two different target processor naming schemes used in the crossbar input file and in the download input file into one. Now labels of the form P1 through P63 are used in the crossbar input file and the download input file.

# Table of Contents

# List of Tables

# List of Figures

# 1. Scope

## 1.1 Identification

This Software Programmer's manual applies to the Georgia Tech Parallel Function Processor (PFP), Georgia Tech part number CERL002-0757-000.0. The Parallel Function Processor (PFP) hardware and software are partitioned into the following two categories; host and target.

The host computer hardware consists of an Intel 310, terminal, and printer. The target computer hardware is the main PFP system unit consisting of thirty-two processing elements, a 16x16 crossbar, the crossbar sequencer, and associated interconnections. The processing elements are generally single board computers (SBC), and are referred to as target processors. The PFP can be upgraded to two clusters of thirty-two processing elements, two crossbars, and two crossbar sequencers.

The system software is divided into two sections; host software and target software. Software which executes on the host and communicates with the PFP during a simulation is referred to as a host server program. Target software, which is executed on the PFP, is divided into three sections. Programs executed on target processors are called processor code. The microcode loaded into the sequencer memory is called sequencer code. The microcode loaded into the crossbar memory is called crossbar code. All target software is written and compiled at the host, then downloaded to the PFP for execution.

## 1.2 System Overview

The purpose of the Parallel Function Processor is to solve systems of differential equations in real-time via the parallel architecture of the machine. The crossbar communication structure between processors facilitates this by allowing a sequence of flexible and dynamic communication events to occur. Each processor can be assigned one or more differential equations (states) to solve. Using the crossbar, state information can be communicated between the processors simultaneously so that the solution is calculated fast and accurately. Not only is the PFP well suited for solving systems of differential equations it is also appropriate for many other programs that can be partitioned into modules, where all communication paths and data transfer lengths are known in advance.

## 1.3 Document Overview

This document contains the information for a programmer to understand and program the Parallel Function Processor. Information on languages, syntax, and memory limits will be presented. Additional information on how to use existing system software is discussed.

## 2. Referenced documents

The following documents contain information which is useful in unerstanding the PFP hardware and software. These documents should be consulted for additional details on specific issues.

Intel iRMXII Reference Manuals Volume 1-7
Intel 80286 SBC Hardware Reference Manual
Intel 80386 SBC Hardware Reference Manual
Intel 214 Disk Controller Hardware Reference Manual
Intel iRMXII C Software Reference Manual
Intel iRMXII FORTRAN Software Reference Manual
Intel iRMXII PASCAL Software Reference Manual
Intel iRMXII PLM Software Reference Manual
Georgia Tech PFP Technical Data Package
PFP Hardware Operation Manual
Georgia Tech GT-FPP/3 Hardware and Software Manuals

# 3. Software Programming Environment

## 3.1 Equipment Configuration

### 3.1.1 Intel 310 Host Computer

The host computer consists of an Intel 310, Multibus I based computer, a 40 Mbyte fixed disk drive, a 360K floppy disk drive, and a terminal. The Intel 310 computer is a 80286 based computer with 2MB memory running the iRMXII operating system. The host serves as the platform for software development and as the interface to the PFP. A Multibus I repeater system is used to interconnect the host to the PFP and all communication between the host and each processing element is accomplished via the Multibus repeater system.

### 3.1.2 PFP Target Computer

The PFP consists of 32 processing elements, one crossbar, and one sequencer and can be upgraded to 64 processing elements, two crossbars, and two sequencers. The host can access all of these through the Multibus I repeater system. The processing elements are usually single board computers but can be other items such as an array interconnect board or an analog I/O board. The crossbar is the dynamic switch that allows flexible data communications between the processing elements. The sequencer is the device that controls the crossbar switching based on an apriori sequence of instructions describing the set of communication patterns to be performed between the processing elements during a simulation. Each processing element has a Multibus I interface and a sequencer/crossbar interface. Interfacing to the PFP can be done either through the Multibus I port and/or a crossbar/sequencer port.

## 3.2 Operational Information

### 3.2.1 Intel 310 Host Computer

For more in depth coverage of the features of the host refer to the Intel iSBC286/12 Hardware Reference Manual and the Parallel Function Processor Operation Manual.

### 3.2.2 PFP Target Computer

The PFP configuration uses several types of processing elements. The Intel 286/12 and 386/12 are commercially available single board computers. Other processing elements have been developed by Georgia Tech as special purpose, high performance processors.

#### 3.2.2.1 Intel Single Board Computer

The 80286 single board computer contains 1 Megabyte of memory which is accessible via the Multibus repeater system. The 286/12 uses an 80287 co-processor. Clock frequency is 8MHz. Refer to the Intel iSBC286/12 Hardware Reference Manual for detailed information.

The 80386 single board computer contains 1 Megabyte of memory which is accessible via the Multibus repeater system. The 386/12 uses an 80387 co-processor. Clock frequency is 20MHz. Refer to the Intel iSBC386/12 Hardware Reference Manual for detailed information.

The GT-FPP/3 single board computer contains 4K of 96 bit wide instruction memory and 2K of 32 bit wide data memory. All instruction memory is accessible via the Multibus repeater system. The FPP uses an AMD 29C325 processor. Clock frequency is 10MHz. The GT-FPP/3 has a throughput of 8 MFlops. Refer to the Georgia Tech GT-FPP/3 Hardware and Software Reference Manuals for detailed information.

#### 3.2.2.2 Sequencer and Crossbar

The sequencer and crossbar work in conjunction with the processing elements to yield inter-processor communication. A sequence of communication patterns described in a crossbar input file is used to generate microcode for the crossbar and sequencer. The crossbar microcode defines which paths on the crossbar are connected and the sequencer microcode selects which processing elements will be involved during a particular communication cycle, waits for the appropriate status flags, generates the data transfer signals, and then advances to the next communication cycle. Refer to the Georgia Tech GT-SEQ/2 Sequencer Design section and to the GT-XB/2 Crossbar Design section of the Georgia Tech PFP Technical Data Package.

## 3.3 Compiling, Binding and Building

The PFP has up to two clusters of thirty-two processing elements (usually processors). The thirty-two processing elements in a single cluster communicate with each other over one 16x16 crossbar. The Multibus I bus repeater system is used by the host to communicate with the PFP. The host computer is used to generate the object code for the processing

elements. At run-time the host computer performs downloading and starts the target processors.

The basic concepts are:

i) During a simulation each active processing element performs a function (a computer program) and, when necessary, sends data to or receives data from other processing elements over the crossbar.

ii) All processors are downloaded with a program using an input file containing a list of the processors and the programs to be loaded into the processors. Some processing elements like the GT-ADDA/2 analog I/O board do not 'run a program' but have the ability to perform any required communcation and processing.

iii) During a simulation the majority of the communication between processing elements takes place across the crossbar, although inter-processor communication can occur via the host computer.

iv) The desired direction for communication between processing elements must be allowed by all the involved elements. Processors request this through a program statement which controls communication direction (e.g., send, receive). All processing elements involved in a given communication cycle must allow the requested transfer.

v) The crossbar/sequencer combination must know and allow the desired communication; (this control is handled through the network definition file, i.e. NETWORK.TXT.)

After the required programs are written, the following items must be done before 'run time':

1) Compile target program
   **ic286 example.c large**
   **ftn286 example.for large**
   **pas286 example.pas large**
   **plm286 example.plm large**

2) Bind and Build target program
   **submit :PFP:csd/CBLDL( example, example.obj )**
   **submit :PFP:csd/FORBLDL( example, example.obj )**
   **submit :PFP:csd/PASBLDL( example, example.obj )**
   **submit :PFP:csd/PLMBLDL( example, example.obj )**

3) Compile network code in the file network.txt (if using the crossbar and sequencer)
   **submit :PFP:csd/XBC(network.txt)**

**Note:** Binding and Building the target program code requires several long commands. These commands have been placed in command files for the different languages. The commands used to invoke these command files are listed above. The listings for these command files can be found in Appendix B.

At run time the following must be performed:

1) reset the PFP system

**-reset**

2) load the target processors (and network if used)

**-download process.txt**

3) start the target processors (and network if used)

**-start process.txt**

4) start the host io server program

**-ioserve process.txt <timeout in second(s)>**

**Note:** The above items 1-4 will generally be contained in a command file or makefile. A discussion of these utilities can be found in Appendix G.

### 3.3.1 Intel 310 Host Computer

#### 3.3.1.1 Compiling

For a comprehensive approach to compiling a program refer to the appropriate Intel language manual listed in section 2 of this document. Some brief examples are:

C:

**- ic286 file_name.c debug large**

FORTRAN:

**- ftn286 file_name.for debug large**

PASCAL:

**- pas286 file_name.pas debug large**

PLM:

**- plm286 file_name.plm debug large**

**Note:** No extensions are assumed for the compilers (i.e., if a FORTRAN file was named 'test.for', 'ftn286 test.for' NOT 'ftn286 test' is required. However the output file for the compiler always replaces the ext(ension) of the input file name with '.obj' (e.g., 'ftn286 test.for' outputs 'test.obj'). The compilers also generate a compiled listing with the extension '.lst'. See the appropriate compiler manual for controls that affect the list file and object file generation.

#### 3.3.1.2 Binding

For a comprehensive look at binding refer to the iAPX 286 Utilities Users' Guide. The following is a brief summary of the items that need to be addressed.

In the iRMXII environment all references to routines that are not defined within the code written by the user have to be resolved at bind time. Although this is not uncommon, in addition to any user defined external references there are multiple language dependent and independent libraries that have to be identified and bound into the program such as co-processor libraries. Another area to be addressed is that of making the bound code into a host executable image. This is accomplished by the RC option of the bind command. Example:

**- bnd286 test.obj, fortran.lib object(test) rc**

This binds test.obj and library fortran.lib to an output file named 'test' and the RC option makes it a host executable image file.

#### 3.3.1.3 Running

To run a file that has been compiled and linked simply type the executable image file name

**- test**

and the program will run. Refer to Appendices A and B for a brief overview of I/O control at the iRMXII level.

### 3.3.2 PFP Target Computer

#### 3.3.2.1 Target Processor

##### 3.3.2.1.1 Compiling

See as section 3.3.1.1

3.3.2.1.2 Binding and Building

For a comprehensive look at binding refer to the iAPX 286 Utilities Users' Guide. For a comprehensive look at building refer to the iAPX 286 System Builder Users' Guide. The following will give a brief summary of the items that need to be addressed.

In the iRMXII environment all references to routines that are not defined within the code written by the user have to be resolved at link time. Although this is not uncommon, in addition to any user defined external references there are multiple language dependent and independent libraries that have to be identified and bound into the program such as co-processor libraries. Example:

**- bnd286 test.obj, fortran.lib object(test.lnk) noload**

This binds test.obj and library fortran.lib to an output file named 'test.lnk'.

The binder output file 'test.lnk' now needs to be built with the bld286 utility. The builder utility assigns absolute addresses for run time.

Appendix B contains listings of several command files that can be used to bind and build programs as well as examples on how to use these command files.

3.3.2.1.3 Downloading

To download a boot loadable program, the utility download is used and is invoked by:

**-download PROCESS.TXT**

where PROCESS.TXT contains lines which consist of four fields separated by blanks: (1) name {p00-p31, p32-p63, crossbar-crossbar2, sequencer-sequencer2}, (2) program filename, (3) input filename (or <NULL> if input not required) and (4) output filename (or <NULL> of output not required). A line with '#' in column 1 is considered a comment and will be ignored. Excluding comments, every line must include each field.

```
# network definition
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
```

3.3.2.1.4 Running

To start the target processors, the utility start is invoked by:

**- start PROCESS.TXT**

#### 3.3.2.1.5 INPUT/OUTPUT

A host program may be executed in order to serve as the interface to host I/O facilities. The host program must read in the PROCESS.TXT file.

**Note:** The language for the host needs to be able to support a structured type of memory access. The preferred language for the host is currently C.

The host program must also handle any input or output needed from target processors during the simulation.

### 3.3.2.2 Crossbar and Sequencer

The crossbar and sequencer microcode are both generated by the crossbar compiler. The input to the crossbar compiler is a crossbar definition file (usually named NETWORK.TXT), which describes the network communication definition statements.

#### 3.3.2.2.1 Compiling

Assume a file named NETWORK.TXT has been created and contains network communication definition statements. The file would be compiled by entering 'xbc' at the iRMXII prompt and then responding with NETWORK.TXT for the input file name and then 'n' for the two follow up questions.

- **xbc**

  Enter the communication file name - NETWORK.TXT

  Do you want setup maps - **n** (generates SETUP.DAT)

  Do you want address maps - **n** (generates ADDRESS.DAT)

If an 'n' is given in response to the latter two questions, the crossbar compiler (xbc) generates two files named sequencer.bl and crossbar.bl. The former is the microcode for the sequencer and the latter is the microcode for the crossbar in boot loadable format. If a 'y' is given in response to the latter two questions the files setup.dat and address.dat are generated and can be used for debugging.

The normal output to the terminal for the crossbar compiler is to list a sequence denoting which cycle is being processed. Errors are written to an 'error.dat' file.

Note: A command file has been provided which will execute the crossbar compiler and answer both of the questions with a 'n'. This command file is executed by:

**- submit :PFP:/csd/XBC( network.txt )**

3.3.2.2.2 Binding and Building

No binding and building is required after compiling crossbar and sequencer code. The file is ready to download.

3.3.2.2.3 Downloading

To download a boot loadable program, the utility download is used and is invoked by:

**-download PROCESS.TXT**

where PROCESS.TXT contains lines which consist of four fields separated by blanks: (1) name {p00-p31, p32-p63, crossbar-crossbar2, sequencer-sequencer2}, (2) program filename, (3) input filename (or <NULL> if input not required) and (4) output filename (or <NULL> of output not required). A line with '#' in column 1 is considered a comment and will be ignored. Excluding comments, every line must include each field.

```
# network 1 definition
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
```

3.3.2.2.4 Running

To start the crossbar and sequencer, the utility start is invoked by:

**- start PROCESS.TXT**

# 4. Programming Information

## 4.1 Host Computer

The Intel 310 host is a commercially available computer with special functions and procedures written by Georgia Tech to enable it to communicate with the PFP. Refer to the iRMXII manual set, the iSBC286/12 hardware reference manual, and the appropriate language manual for comprehensive information. Appendices C, D, and E list the special functions and procedures available to communicate with the PFP. The following paragraphs give a general discussion of the more important topics.

### 4.1.1 Programming Features

Refer to the appropriate Intel language reference manual listed in Section 2 of this document for the data types supported by a particular language.

### 4.1.2 Programming Instructions

See the appropriate Intel language manual(s) listed in Section 2 of this document.

### 4.1.3 Input and Output Control Programming

Two serial ports and one parallel I/O port are available on the single board computer module in the Intel 310. The terminal is connected to one serial port and a printer may be connected to the parallel port. Additionally, an iSBX expansion port is available on the single board computer. The host computer is interfaced to the PFP through the Multibus. This interfacing is accomplished via a repeater scheme where the host computer utilizes a 'host repeater' board and the PFP's Multibus card cages use 'slave repeater' boards. Low-level program routines access this repeater system in order to pass data between the host and target computers. All standard I/O routines such as reads, writes, and file manipulation utilities are available and can be utilized.

### 4.1.4 Additional or Special Techniques

The low-level utilities for communicating between the host and the PFP are addressed in this section.

The repeater system that is used to communicate between the host and the PFP is based on a memory window scheme. In order to access more than the 16 Megabyte address space of

the Multibus I specification, a paging scheme was developed and implemented on the PFP. The paging scheme has a window size of 1MB. The use of a window is accessed at a specific 1MB boundary, but all of the window need not be filled. After the repeater is 'turned on', the memory of the target processor at that window appears and is accessed as such via the input/output routines listed in Appendices C, D, E and F. Low level routines input_buffer/output_buffer are used on both the target and the host in order to achieve a data transfer. After the required transfer is completed the repeater is 'turned off'.

This procedure of turning the repeater on, communicating, and then turning the repeater off is much like the process of making a telephone call, trading the required information, and then hanging up. Each window represents one target processor. Before starting the next communication sequence with another target processor it is required to complete the above sequence by turning the repeater off.

Listings of special I/O functions and procedures are given in Appendix C.

#### 4.1.5 Programming Examples

All communication between the host and the target processors can be accomplished with the IOSERVE utility eliminating the need for development of communication software for the host. For examples of programming the host to perform other tasks such as data analysis, refer to the appropriate intel language reference manual(s) listed in section 2 of this document.

#### 4.1.6 Error Detection and Diagnostic Features

Compilation and binding errors are written to the terminal and to the appropriate list file (.lst for compile or .mp1 for bind). Run time errors are displayed on the terminal. Refer to the Intel iRMXII manuals and Intel language manuals for explanations of the errors.

### 4.2 Target Computer

#### 4.2.1 Target Processors

The Intel iSBC286/12 is a commercially available single board computer. Refer to the iRMXII manual set, the iSBC286/12 hardware reference manual, and the appropriate language manual for comprehensive information. The following paragraphs give a general discussion of the more prominent information.

#### 4.2.1.1 Programming Features

Refer to the appropriate Intel language reference manual listed in Section 2 of this document for the data types supported by a particular language. See Appendices C, D, E and F for the I/O routines supported for C, FORTRAN, PASCAL and PLM, respectively.

#### 4.2.1.2 Programming Instructions

See the appropriate Intel language manual(s) listed in Section 2 of this document.

#### 4.2.1.3 Input and Output Control Programming

Two serial ports and one parallel port are available on each processor for connection to external devices. One of the serial ports can be utilized by a set of instructions given in Appendices C, D, E and F for C, FORTRAN, PASCAL and PLM, respectively. Additionally, an iSBX expansion port is available on the board. The target processors have an interface with both the host computer via the Multibus and with the crossbar network through the iSBX port. No standard I/O routines such as reads, writes, and file manipulation are available on the target processors. Only the low-level routines to communicate to the host via the Multibus and the crossbar via the iSBX port are supported.

#### 4.2.1.4 Additional or Special Techniques

The low-level utilities for communicating between the host and the PFP are addressed in this section. See section 4.1.4 for more background on the repeater system for Multibus communications between the target processors and the host.

The target processors can communicate to either the host or to another target processor. Communication with the host is accomplished through the multibus repeater system. Communication to the host across the Multibus can only be performed when the host has accessed a specific processing element (i.e., when the host has turned on the memory page to the element.) In order to transfer data the target processor issues a send for every receive issued by the host, and a receive for every send issued by the host.

The target processors communicate with each other by issuing sends and receives via the iSBX port to the crossbar. The other processors involved in a crossbar communication cycle are required to accept or generate data appropriately. Additionally, as covered in section 4.2.2, the crossbar and sequencer will need to be programmed to accommodate these communication patterns.

#### 4.2.1.5 Programming Examples

Refer to Appendix C, D, E and F for a complete example for each language.

#### 4.2.1.6 Error Detection and Diagnostic Features

Compilation, binding, and building errors are written to the terminal and to the appropriate listing file (.lst for compile, .mp1 for bind, or .mp2 for build). Refer to the Intel iRMXII language manuals for explanations. Run time errors will be determined by the correctness and the completeness of an execution sequence.

### 4.2.2 Crossbar and Sequencer

The crossbar and sequencer work as a unified system. The crossbar compiler (xbc) generates microcode for the crossbar and the sequencer from an input file that describes the communication cycles and patterns to be implemented during the execution of a multi-processor program. The sequencer controls or as the name implies, sequences the crossbar through the defined data path connections described by the input file. Each crossbar and sequencer combination supports 32 processing elements and allows for multiple sets of communications to occur at the same time. A communication cycle is one processing element transferring data to one or more processing elements.

#### 4.2.2.1 Programming Features

The communication paths for the crossbar are 16 bits wide. Thus all communication between processors is performed as sets of 16 bit transfers (e.g., a 16 bit integer is transferred in one transfer cycle, a 32 bit floating point number is transferred in two cycles.)

Single instructions allow the transfer of data from one processing element to one or more processing elements. The sequencer and compiler combination supports two constructs or flow control statements: (i) CYCLE and (ii) LOOP. The CYCLE construct allows for the grouping of one or more single instructions into one simultaneous communication. The CYCLE construct allows for parallel communications to occur. The LOOP construct is used to define a set of CYCLEs that are to be repeated indefinitely.

An example of this would be a simulation that requires some initial data transfer to initialize all the state variables and then to begin an integration routine where a set of variables needs to be communicated during each integration step. The initial data transfer requires a CYCLE construct which is executed once. The integration requires a CYCLE construct which is executed repeatedly using the LOOP construct.

Comments are opened by a '[' and closed by a ']' and cannot be nested. The CYCLE construct groups the single instructions between the current CYCLE statement and the next

CYCLE statement. By definition the CYCLE constructs cannot be nested. The LOOP construct can only be used once per input file. LOOP lumps all the CYCLE statements that follow it into one big loop (i.e. all statements between the LOOP statement and the end of the file are grouped together and are repeated indefinitely.)

#### 4.2.2.2 Programming Instructions

The only one instruction is the transfer instruction which has the following syntax:

$P_i,P_k,\cdots,P_j := P_l[.n][;]$

where processor $P_l$ is transferring data to the set of processors $P_i,P_k,\cdots,P_j$. The number of 16 bit transfers is controlled by the .n option where n is an integer. The ';' is an optional line terminator. If the .n option is omitted the default is n=1. If a 32 bit value is to be transferred, the n is replaced by a 2 (e.g., := $P_l$.2). The set of processors $P_i,P_k,\cdots,P_j$ does not have to be monotonically increasing or decreasing, but for clarity it is helpful if the sequence is monotonic. All subscripts for the P(rocessor) numbers are in the set [0,63]. The processor id appearing on the right side of the transfer equation cannot appear on the left. It also follows that a processor id cannot appear more than once during a cycle.

#### 4.2.2.3 Input and Output Control Programming

The communication between the sequencer and the host is accomplished with the same repeater scheme used for the target processors. All work is done by the host and the sequencer is used as a bank of memory until the I/O start command is issued. At this time the sequencer begins sequencing the communication activities. The crossbar is accessed through the sequencer and is controlled as a memory bank, similar to the sequencer, via the repeater system.

#### 4.2.2.4 Additional or Special Techniques

Since the communication patterns have to be determined apriori, all possible data transfers between processing elements must be described in the crossbar input file. This requires that the processing elements generate and accept transfers during each loop of the LOOP code. The processing elements will know or determine which datums are valid and which are not valid.

#### 4.2.2.5 Programming Examples

The following is an example crossbar definition file.

Figure 1. Example Crossbar File

```
[ Example Crossbar ]

        CYCLE                                       [ initialize ]
                p4,  p5,  p9  := p15.2;
                p2 := p3.2;

LOOP                                                [ main loop ]
        CYCLE
                p1,  p2,  p3  := p0;

        CYCLE
                p4,  p5,  p9  := p15.2;
                p0,  p2,  p3  := p1;

        CYCLE
                p0,  p1,  p3  := p2;

        CYCLE
                p0,  p1,  p2  := p3;
```

#### 4.2.2.6 Error Detection and Diagnostic Features

The normal output to the terminal for the crossbar compiler is a numerical sequence denoting each cycle as it is being processed. Compilation errors are written to an 'error.dat' file as they are encountered during compilation and are self-explanatory. Some common errors are: (i) using periods instead of commas, (ii) use of the same processor id on both sides of the ':=', and (iii) the same processor id used more than once during a CYCLE. Run time errors will be determined by the correctness and the completeness of an execution sequence.

## 5. Notes

Programming in a parallel environment requires considerations not encountered in a serial environment. Since there are multiple processes occurring simultaneously, coordination of communication and computation for each process is required. Data dependencies between processes have to be such that one processor is not expecting data from another processor which in turn is expecting data from the former. This state is referred to as deadlock. Deadlock can also happen between more than two processes.

Data to be transferred from one process to another requires a communication path. When the programmer is scheduling the inter-processor communication (i.e., writing the crossbar code) attention has to be given to determine if the desired number of transfers can be performed during one cycle. Depending on the number of transfers already scheduled in a specific cycle, the addition of another data transfer may need to be delayed until the next available cycle, unless one of the currently scheduled data exchanges can be moved.

Communication between multiple target processors, or between target processors and the host, have to be coordinated so that each process sending information has a process to correctly receive each datum being sent (and vice versa). These communications are matched according to data type (real, integer, etc.). Inter-processor communications also requires matching sequencer and crossbar code (the inter-processor communication file) in order for target processors to communicate, in addition to the sends and receives on the processors.

Sending control information to processes, and debugging in a parallel environment, involves the receiving and sometimes the sending of messages to the individual processes during a simulation. Each of these messages must be tagged or labeled appropriately so a thorough understanding of the information is easily achieved. On the PFP, all interaction with the processes by the operator is accomplished via the host. Interaction implies sending a (data) message to the processor or receiving a (data) message from the processor. As an example, if a process needs to send out a message to the operator, the process sends the message to the host and then the host either displays it at the terminal or stores it in a file. Other forms of displaying information by the processes can be through analog I/O or through the control of status LEDs on the individual processors. This process is analogous to having to perform all I/O for a standard program in the main program, with information being generated in or transmitted to subprograms, as opposed to being able to do at will the desired reads and writes in the subprograms.

Appendices C, D, E and F contain the message passing information for both processor to host communication and processor to processor communication for C, FORTRAN, PASCAL and PLM respectively.

# 6. Appendices

# 6.1 Appendix A

## Summary of iRMXII Commands

Operating system : iRMXII

Intel iRMXII is not case sensitive, except in the case of the account password.

Some useful commands:

**- submit filename**

runs a command file with name filename.csd

A command file is a text file with a series of iRMXII commands that are to be executed together repetitively. The .csd file is very similar to the .bat file in DOS and the .com file in VAX/VMS.

**- submit :PFP:csd/xbc(filename)**

runs command file xbc with parameter 'filename'

**- submit filename to :LP:**

runs command file filename and sends output to :LP:

**- submit filename to filename.out echo**

runs command file and sends output to filename.out and echos it to the screen

**- attachfile path** (or **cd path**)

changes directory to path (if it exits)

Examples this and other uses of cd:

**cd :HOME:**

**cd /pfp/test**

**cd /** (moves to root directory)

**cd ^** (moves to parent directory) )

**cd :sd:** (moves to root directory)

**cd $ as :a_name:** (assigns :a_name: to the current directory)

**cd :a_name:** (moves to directory specified

by the logical :a_name:)

**cd :HOME:SRC/EXOSIM to :EXOSIM:**

**cd :EXOSIM:**

**- createdir subdir**

makes directory as specified by subdir

**- copy filename(s)**

displays the contents of said file on the screen

**- copy [path]file1 to [path]file2**

copy file1 to file2

**- copy file(s) to :lp:**

prints file to printer

**- copy file(s) to :$:**

copies file(s) to current directory

**- dir (or dir :$: or dir $)**

directory

**- dir :PFP:**

**- dir /system**

**- dir $ for *.obj**

directory of files ending in '.obj' in the current directory :$:

**- dir $ to :LP: for *.dat**

sends directory of :$: to printer :LP:

**- attachdevice wmf0 as :f:**

allows the floppy in the floppy drive to be used as device :f: See detachdevice :f:

**- detachdevice :f:**

removes the floppy from use - must be used before removing floppy!

**- attachdevice wta0 as :tape: physical**

allows the tape in the drive to be used as :tape:

**- detachdevice :t:**

removes the tape from use

**- delete file(s)**

delete file(s) and empty directories

**- permit file_name(s) drau user=1**

sets common file access privilege to 'file_name'

(e.g., **- permit RK4.for drau user=1**

**- permit test/* drau user=1**

**- permit * drau user=1** )

**- path**

displays current directory path (e.g., /user/pfp/test)

**- ^R**

recalls previous commands (can not go forward list)

**- ^C**

cancels current command

**- ^W**

when entered before doing a copy, this command allows paging by typing a ^W

**- backup :sd: to :tape:**

backs up the files on disk :sd: to tape :tape:

**- restore** : see the iRMXII manuals.

- **super** : prompts for password to be super user

- **alias** : the alias command allows the user to alias iRMX commands to those more familiar to the user. Aliases commonly used should be entered into the r'?'logon file in the users 'prog' (:prog:) directory. Examples

    alias ls = dir

    alias ls = dir $ 1

    alias cd = attachfile

These aliases also allow parameters to be entered on the command line. These parameters will be inserted into the command as it is executed.

An amperstand '&' is used at the end of a line to continue on to the next line.

- **aedit filename.ext**

    invokes the AEDIT fullscreen editor

    - use **TAB** to view command bar at bottom of screen

    - **ESC** completes an input sequence on most commands

    - **^C** aborts a commmand

    - use arrow keys

    - **HOME** moves (page, BOL, EOL) w.r.t. the last arrow movement

    - **i**(nsert)

    - **d**(elete) move cursor **d**(elete) : deletes region

    - **b**(uffer) move cursor **b**(uffer) : buffers region

    - the current version of the file is saved under the name filename.bak

## 6.2 Appendix B

# Compiling, Binding, and Building

Table B-1 Filename Conventions

| File Extension | File Usage |
|---|---|
| *.CSD | command file |
| *.INC | include file |
| *.C | C source file |
| *.FOR | FORTRAN source file |
| *.PAS | PASCAL source file |
| *.PLM | PLM source file |
| *.TXT | text file |
| *.LST | compiler listing file |
| *.OBJ | compiler object file |
| *.MP1 | binder listing file |
| *.LNK | binder object file |
| *.MP2 | builder listing file |
| *.BL | builder object file |
| *.BLD | builder command file |
| *.MAP | map listing file |
| *.LIB | library object file |

The following is a command file (CBNDL.CSD) for binding a Intel iRMXII C program for the host processor.

```
Example:
   ic286 example.c debug large
   submit :PFP:csd/cbndl( example, example.obj, debug )
```

Figure B-1. Host Processor - CBNDL.CSD

```
;
; cbndl( <name>, <object>, <bnd option> )
;
bnd286 &
:LIB:ic286/cstart2l.obj, &
%1, &
:PFP:lib/host.lib, &
:LIB:ic286/lib2l.lib, &
:LIB:ndp287/cel287.lib, &
:LIB:ndp287/80287.lib,
/rmx286/lib/rmxifl.lib, &
/rmx286/lib/udiifl.lib &
object(%0) ss(stack(+1000h)) &
rc(dm(01000h,fffffh)) fastload %2
```

The following is a command file (CBLDL.CSD) for binding and building a Intel iRMXII C program for the target processors.

```
Example:
   ic286 example.c large
   submit :PFP:csd/cbldl( example, example.obj )
```

Figure B-2. Target processor - CBLDL.CSD

```
;
; cbldl( <name>, <object>, <bnd option>, <bld option> )
;
bnd286 &
:PFP:lib/cstartl.lnk, &
%1, &
:PFP:lib/interrupt.obj, &
:PFP:lib/target.lib, &
:LIB:ic286/clib2l.lib, &
:LIB:ic286/cfloat2l.lib, &
:LIB:ndp287/cel287.lib, &
:LIB:ndp287/80287.lib &
object(%0.lnk) ss(stack(+1000H)) &
name(maintask) noload %2

bld286 &
:PFP:lib/inittask.obj, &
%0.lnk &
object(%0.bl) buildfile(:PFP:lib/target.bld) bootload %3

delete %0.lnk
```

The following is a command file (FORBNDL.CSD) for binding a Intel iRMXII FORTRAN program for the host processor.

```
Example:
   ftn286 example.for debug large
   submit :PFP:csd/forbndl( example, example.obj, debug )
```

Figure B-3. Host Processor - FORBNDL.CSD

```
;
; forbndl( <name>, <object>, <bnd option> )
;
bnd286 &
%1, &
:LIB:ftn286/f286r0.lib, &
:LIB:ftn286/f286r1.lib, &
:LIB:ftn286/f286r2.lib, &
:LIB:ftn286/f286r3.lib, &
:LIB:ftn286/f286r4.lib, &
:LIB:ndp287/cel287.lib, &
:LIB:ndp287/80287.lib, &
/rmx286/lib/rmxifl.lib, &
/rmx286/lib/udiifl.lib &
object(%0) ss(stack(+1000h)) &
rc(dm(01000h,fffffh)) fastload %2
```

The following is a command file (FORBLDL.CSD) for binding and building a Intel iRMXII FORTRAN program for the target processors.

```
Example:
   ftn286 example.for large
   submit :PFP:csd/forbldl( example, example.obj )
```

Figure B-4 Target Processor - FORBLDL.CSD

```
;
; forbldl( <name>, <object>, <bnd option>, <bld option> )
;
bnd286 &
%1, &
:PFP:lib/interrupt.obj, &
:PFP:lib/target.lib, &
:LIB:ftn286/f286r0.lib, &
:LIB:ftn286/f286r1.lib, &
:LIB:ftn286/f286r2.lib, &
:LIB:ftn286/rtn286.lib, &
:LIB:ndp287/cel287.lib, &
:LIB:ndp287/80287.lib &
object(%0.lnk) ss(stack(+1000H)) &
name(maintask) noload %2

bld286 &
:PFP:lib/inittask.obj, &
%0.lnk &
object(%0.bl) buildfile(:PFP:lib/target.bld) bootload %3

delete %0.lnk
```

The following is a command file (PASBNDL.CSD) for binding a Intel iRMXII PASCAL program for the host processor.

```
Example:
   pas286 example.pas debug large
   submit :PFP:csd/pasbndl( example, example.obj, debug )
```

Figure B-5. Host Processor - PASBNDL.CSD

```
;
; pasbndl( <name>, <object>, <bnd option> )
;
bnd286 &
%1, &
:LIB:pas286/p286r0.lib, &
:LIB:pas286/p286r1.lib, &
:LIB:pas286/p286r2.lib, &
:LIB:pas286/p286r3.lib, &
:LIB:ndp287/cel287.lib, &
:LIB:ndp287/80287.lib, &
/rmx286/lib/rmxifl.lib, &
/rmx286/lib/udiifl.lib &
object(%0) ss(stack(+1000h)) &
rc(dm(01000h,fffffh)) fastload %2
```

The following is a command file (PASBLDL.CSD) for binding and building a Intel iRMXII PASCAL program for the target processors.

```
Example:
   pas286 example.pas large
   submit :PFP:csd/pasbldl( example, example.obj )
```

Figure B-6 Target Processor - PASBLDL.CSD

```
;
; pasbldl( <name>, <object>, <bnd option>, <bld option> )
;
bnd286 &
%1, &
:PFP:lib/interrupt.obj, &
:PFP:lib/target.lib, &
:LIB:pas286/p286r0.lib, &
:LIB:pas286/p286r1.lib, &
:LIB:pas286/rtn286.lib, &
:LIB:ndp287/cel287.lib, &
:LIB:ndp287/80287.lib &
object(%0.lnk) ss(stack(+1000H)) &
name(maintask) noload %2

bld286 &
:PFP:lib/inittask.obj, &
%0.lnk &
object(%0.bl) buildfile(:PFP:lib/target.bld) bootload %3

delete %0.lnk
```

The following is a command file (PLMBNDL.CSD) for binding a Intel iRMXII PLM program for the host processor.

```
Example:
   plm286 example.plm debug large
   submit :PFP:csd/plmbndl( example, example.obj, debug )
```

Figure B-7. Host Processor - PLMBNDL.CSD

```
;
; plmbndl( <name>, <object>, <bnd option> )
;
bnd286 &
%1, &
:LIB:plm286/plm286.lib, &
/rmx286/lib/rmxifl.lib, &
/rmx286/lib/udiifl.lib &
object(%0) ss(stack(+1000h)) &
rc(dm(01000h,fffffh)) fastload %2
```

The following is a command file (PLMBLDL.CSD) for binding and building a Intel iRMXII PLM program for the target processors.

```
Example:
   plm286 example.plm large
   submit :PFP:csd/plmbldl( example, example.obj )
```

Figure B-8 Target Processor - PLMBLDL.CSD

```
;
; plmbldl( <name>, <object>, <bnd option>, <bld option> )
;
bnd286 &
%1, &
:PFP:lib/interrupt.obj, &
:PFP:lib/target.lib, &
:LIB:plm286/plm286.lib, &
:LIB:ndp287/cel287.lib, &
:LIB:ndp287/80287.lib &
object(%0.lnk) ss(stack(+1000H)) &
name(maintask) noload %2

bld286 &
:PFP:lib/inittask.obj, &
%0.lnk &
object(%0.bl) buildfile(:PFP:lib/target.bld) bootload %3

delete %0.lnk
```

Figure B-9 Target Build File - TARGET.BLD

```
BUILDFILE;

SEGMENT
    *SEGMENTS (DPL=0),
    STARTUP_DATA (BASE=001000H),
    STARTUP_CODE (BASE=001010H);

TABLE
    GDT (LOCATION=GDT_PTR);

GATE
    INTERRUPT_00_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_00),
    INTERRUPT_01_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_01),
    INTERRUPT_02_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_02),
    INTERRUPT_03_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_03),
    INTERRUPT_04_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_04),
    INTERRUPT_05_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_05),
    INTERRUPT_06_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_06),
    INTERRUPT_07_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_07),
    INTERRUPT_08_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_08),
    INTERRUPT_09_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_09),
    INTERRUPT_10_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_10),
    INTERRUPT_11_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_11),
    INTERRUPT_12_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_12),
    INTERRUPT_13_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_13),
    INTERRUPT_14_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_14),
    INTERRUPT_15_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_15),
    INTERRUPT_16_GATE (INTERRUPT,DPL=0,ENTRY=INTERRUPT_16);

TABLE
    IDT
    (
            LOCATION=IDT_PTR,
            ENTRY=
                    (
                    00:INTERRUPT_00_GATE,
```

```
                    01:INTERRUPT_01_GATE,
                    02:INTERRUPT_02_GATE,
                    03:INTERRUPT_03_GATE,
                    04:INTERRUPT_04_GATE,
                    05:INTERRUPT_05_GATE,
                    06:INTERRUPT_06_GATE,
                    07:INTERRUPT_07_GATE,
                    08:INTERRUPT_08_GATE,
                    09:INTERRUPT_09_GATE,
                    10:INTERRUPT_10_GATE,
                    11:INTERRUPT_11_GATE,
                    12:INTERRUPT_12_GATE,
                    13:INTERRUPT_13_GATE,
                    14:INTERRUPT_14_GATE,
                    15:INTERRUPT_15_GATE,
                    16:INTERRUPT_16_GATE
                    )
    );

TASK
    INITTASK (OBJECT=INITTASK),
    MAINTASK (OBJECT=MAINTASK);

MEMORY
(
RESERVE =
    (
    0000000H..0000FFFH,    -- monitor ram
    0100000H..0F7FFFFH,    -- memory not available
    0F80000H..0FFFFFFH     -- monitor rom
    )
);

END
```

## 6.3 Appendix C

## Target Processor C I/O

The C variable types supported by the I/O routines are described in Table C-1

**Table C-1 Supported I/O variable types for C.**

| data_type | alternate data_type | message_type | network |
|---|---|---|---|
| char | CHARACTER_08BIT_type | CHARACTER_08BIT | p1:=p0.1 |
| struct<br>{<br>float r;<br>float i;<br>} | COMPLEX_32BIT_type | COMPLEX_32BIT | p1:=p0.4 |
| struct<br>{<br>double r;<br>double i;<br>} | COMPLEX_64BIT_type | COMPLEX_64BIT | p1:=p0.8 |
| char | LOGICAL_08BIT_type | LOGICAL_08BIT | p1:=p0.1 |
| short | LOGICAL_16BIT_type | LOGICAL_16BIT | p1:=p0.1 |
| long | LOGICAL_32BIT_type | LOGICAL_32BIT | p1:=p0.2 |
| float | REAL_32BIT_type | REAL_32BIT | p1:=p0.2 |
| double | REAL_64BIT_type | REAL_64BIT | p1:=p0.4 |
| signed char | SIGNED_08BIT_type | SIGNED_08BIT | p1:=p0.1 |
| signed short | SIGNED_16BIT_type | SIGNED_16BIT | p1:=p0.1 |
| signed long | SIGNED_32BIT_type | SIGNED_32BIT | p1:=p0.2 |
| unsigned char | UNSIGNED_08BIT_type | UNSIGNED_08BIT | p1:=p0.1 |
| unsigned short | UNSIGNED_16BIT_type | UNSIGNED_16BIT | p1:=p0.1 |
| unsigned long | UNSIGNED_32BIT_type | UNSIGNED_32BIT | p1:=p0.2 |

The Intel iRMXII C compiler supports including files at compile time. As such we can include files that have definitions for seperately compiled functions and procedures. These files usually have the extension '.c' or '.h'.

The procedures used to send and receive data between target processors follow the form:

**send_message_type(&message);** and

**receive_message_type(&message);**

where **message_type** is replaced with an entry from Table C-1 and where message is a scalar variable and is the corresponding data_type or alternate data_type.

The procedures used to send and receive data between the target processors and the host follow the form:

**output_message(message_type,&message,message_size);** and

**input_message(&message_type,&message,&message_size);**

where message_type is replaced with an entry from Table C-1, where message is a scalar or array variable and is the corresponding data_type or alternate data_type, and where message_size is the number of datums in message.

Figure C-1. Target Processor C Example - MAKEFILE

```
cflags = code large optimize(3) \
    searchinclude(:LIB:ic286/,:PFP:include/)

default:    input.bl output.bl crossbar.bl sequencer.bl

input.bl:   input.obj
    submit :PFP:csd/cbldl( input, input.obj )

input.obj: input.c
    ic286 input.c $(cflags)

output.bl: output.obj
    submit :PFP:csd/cbldl( output, output.obj )

output.obj:output.c
    ic286 output.c $(cflags)

crossbar.bl sequencer.bl: network.txt
    submit :PFP:csd/xbc( network.txt )

clean:
    delete *.lst,*.obj,*.mp?,*.bl

run:        input.bl output.bl crossbar.bl sequencer.bl
    reset
    download process.txt
    start process.txt
    ioserve process.txt 2
```

Figure C-2. Target Processor C Example - TARGET.H

```
#define CHARACTER_08BIT 0
#define COMPLEX_32BIT 1
#define COMPLEX_64BIT 2
#define LOGICAL_08BIT 3
#define LOGICAL_16BIT 4
#define LOGICAL_32BIT 5
#define REAL_32BIT 6
#define REAL_64BIT 7
#define SIGNED_08BIT 8
#define SIGNED_16BIT 9
#define SIGNED_32BIT 10
#define UNSIGNED_08BIT 11
#define UNSIGNED_16BIT 12
#define UNSIGNED_32BIT 13

typedef char CHARACTER_08BIT_type;
typedef struct { float r; float i; } COMPLEX_32BIT_type;
typedef struct { double r; double i; } COMPLEX_64BIT_type;
typedef char LOGICAL_08BIT_type;
typedef short LOGICAL_16BIT_type;
typedef long LOGICAL_32BIT_type;
typedef float REAL_32BIT_type;
```

```
typedef double REAL_64BIT_type;
typedef signed char SIGNED_08BIT_type;
typedef signed short SIGNED_16BIT_type;
typedef signed long SIGNED_32BIT_type;
typedef unsigned char UNSIGNED_08BIT_type;
typedef unsigned short UNSIGNED_16BIT_type;
typedef unsigned long UNSIGNED_32BIT_type;

extern void output_nl( void );

extern void input_buffer( void *buffer, signed short buffer_size );
extern void input_message( signed short *message_type,
    void *message, signed short *message_size );
extern unsigned char input_ready( void );

extern void led( signed short state );

extern void output_buffer( void *buffer, signed short buffer_size );
extern void output_message( signed short message_type,
    void *message, signed short message_size );
extern unsigned char output_ready( void );

extern void receive_buffer( void *buffer, signed short buffer_size );
extern void receive_CHARACTER_08BIT( CHARACTER_08BIT_type *buffer );
extern void receive_COMPLEX_32BIT( COMPLEX_32BIT_type *buffer );
extern void receive_COMPLEX_64BIT( COMPLEX_64BIT_type *buffer );
extern void receive_LOGICAL_08BIT( LOGICAL_08BIT_type *buffer );
extern void receive_LOGICAL_16BIT( LOGICAL_16BIT_type *buffer );
extern void receive_LOGICAL_32BIT( LOGICAL_32BIT_type *buffer );
extern void receive_REAL_32BIT( REAL_32BIT_type *buffer );
extern void receive_REAL_64BIT( REAL_64BIT_type *buffer );
extern void receive_SIGNED_08BIT( SIGNED_08BIT_type *buffer );
extern void receive_SIGNED_16BIT( SIGNED_16BIT_type *buffer );
extern void receive_SIGNED_32BIT( SIGNED_32BIT_type *buffer );
extern void receive_UNSIGNED_08BIT( UNSIGNED_08BIT_type *buffer );
extern void receive_UNSIGNED_16BIT( UNSIGNED_16BIT_type *buffer );
extern void receive_UNSIGNED_32BIT( UNSIGNED_32BIT_type *buffer );

extern void send_buffer( void *buffer, signed short buffer_size );
extern void send_CHARACTER_08BIT( CHARACTER_08BIT_type *buffer );
extern void send_COMPLEX_32BIT( COMPLEX_32BIT_type *buffer );
extern void send_COMPLEX_64BIT( COMPLEX_64BIT_type *buffer );
extern void send_LOGICAL_08BIT( LOGICAL_08BIT_type *buffer );
extern void send_LOGICAL_16BIT( LOGICAL_16BIT_type *buffer );
extern void send_LOGICAL_32BIT( LOGICAL_32BIT_type *buffer );
extern void send_REAL_32BIT( REAL_32BIT_type *buffer );
extern void send_REAL_64BIT( REAL_64BIT_type *buffer );
extern void send_SIGNED_08BIT( SIGNED_08BIT_type *buffer );
extern void send_SIGNED_16BIT( SIGNED_16BIT_type *buffer );
extern void send_SIGNED_32BIT( SIGNED_32BIT_type *buffer );
extern void send_UNSIGNED_08BIT( UNSIGNED_08BIT_type *buffer );
extern void send_UNSIGNED_16BIT( UNSIGNED_16BIT_type *buffer );
extern void send_UNSIGNED_32BIT( UNSIGNED_32BIT_type *buffer );

#ifndef TRUE
#define TRUE 1
#endif
#ifndef FALSE
#define FALSE 0
#endif
```

Figure C-3. Target Processor C Example - INPUT.C

```
#include <target.h>

void main( void )
{
    SIGNED_16BIT_type message_type;
    SIGNED_16BIT_type message_size;

    COMPLEX_32BIT_type c32;
    COMPLEX_64BIT_type c64;
    LOGICAL_08BIT_type l08;
    LOGICAL_16BIT_type l16;
    LOGICAL_32BIT_type l32;
```

```
    REAL_32BIT_type r32;
    REAL_64BIT_type r64;

    SIGNED_08BIT_type s08;
    SIGNED_16BIT_type s16;
    SIGNED_32BIT_type s32;

    UNSIGNED_08BIT_type u08;
    UNSIGNED_16BIT_type u16;
    UNSIGNED_32BIT_type u32;

    input_message( &message_type, &c32, &message_size );
    input_message( &message_type, &c64, &message_size );

    input_message( &message_type, &l08, &message_size );
    input_message( &message_type, &l16, &message_size );
    input_message( &message_type, &l32, &message_size );

    input_message( &message_type, &r32, &message_size );
    input_message( &message_type, &r64, &message_size );

    input_message( &message_type, &s08, &message_size );
    input_message( &message_type, &s16, &message_size );
    input_message( &message_type, &s32, &message_size );

    input_message( &message_type, &u08, &message_size );
    input_message( &message_type, &u16, &message_size );
    input_message( &message_type, &u32, &message_size );

    send_COMPLEX_32BIT( &c32 );
    send_COMPLEX_64BIT( &c64 );

    send_LOGICAL_08BIT( &l08 );
    send_LOGICAL_16BIT( &l16 );
    send_LOGICAL_32BIT( &l32 );

    send_REAL_32BIT( &r32 );
    send_REAL_64BIT( &r64 );

    send_SIGNED_08BIT( &s08 );
    send_SIGNED_16BIT( &s16 );
    send_SIGNED_32BIT( &s32 );

    send_UNSIGNED_08BIT( &u08 );
    send_UNSIGNED_16BIT( &u16 );
    send_UNSIGNED_32BIT( &u32 );

} /* main */
```

Figure C-4. Target Processor C Example - OUTPUT.C

```
#include <target.h>

void main( void )
{
    COMPLEX_32BIT_type c32;
    COMPLEX_64BIT_type c64;

    LOGICAL_08BIT_type l08;
    LOGICAL_16BIT_type l16;
    LOGICAL_32BIT_type l32;

    REAL_32BIT_type r32;
    REAL_64BIT_type r64;

    SIGNED_08BIT_type s08;
    SIGNED_16BIT_type s16;
    SIGNED_32BIT_type s32;

    UNSIGNED_08BIT_type u08;
    UNSIGNED_16BIT_type u16;
    UNSIGNED_32BIT_type u32;

    receive_COMPLEX_32BIT( &c32 );
```

```
    receive_COMPLEX_64BIT( &c64 );

    receive_LOGICAL_08BIT( &l08 );
    receive_LOGICAL_16BIT( &l16 );
    receive_LOGICAL_32BIT( &l32 );

    receive_REAL_32BIT( &r32 );
    receive_REAL_64BIT( &r64 );

    receive_SIGNED_08BIT( &s08 );
    receive_SIGNED_16BIT( &s16 );
    receive_SIGNED_32BIT( &s32 );

    receive_UNSIGNED_08BIT( &u08 );
    receive_UNSIGNED_16BIT( &u16 );
    receive_UNSIGNED_32BIT( &u32 );

    output_message( CHARACTER_08BIT, "c32= ", 5 );
    output_message( COMPLEX_32BIT, &c32, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "c64= ", 5 );
    output_message( COMPLEX_64BIT, &c64, 1 );
    output_nl( );

    output_message( CHARACTER_08BIT, "l08= ", 5 );
    output_message( LOGICAL_08BIT, &l08, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "l16= ", 5 );
    output_message( LOGICAL_16BIT, &l16, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "l32= ", 5 );
    output_message( LOGICAL_32BIT, &l32, 1 );
    output_nl( );

    output_message( CHARACTER_08BIT, "r32= ", 5 );
    output_message( REAL_32BIT, &r32, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "r64= ", 5 );
    output_message( REAL_64BIT, &r64, 1 );
    output_nl( );

    output_message( CHARACTER_08BIT, "s08= ", 5 );
    output_message( SIGNED_08BIT, &s08, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "s16= ", 5 );
    output_message( SIGNED_16BIT, &s16, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "s32= ", 5 );
    output_message( SIGNED_32BIT, &s32, 1 );
    output_nl( );

    output_message( CHARACTER_08BIT, "u08= ", 5 );
    output_message( UNSIGNED_08BIT, &u08, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "u16= ", 5 );
    output_message( UNSIGNED_16BIT, &u16, 1 );
    output_nl( );
    output_message( CHARACTER_08BIT, "u32= ", 5 );
    output_message( UNSIGNED_32BIT, &u32, 1 );
    output_nl( );

} /* main */
```

Figure C-5. Target Processor C Example - NETWORK.TXT

```
LOOP;


CYCLE [ 1 ]
 p31 := p15.4; [ c32 ]

CYCLE [ 2 ]
 p31 := p15.8; [ c64 ]
```

```
CYCLE [ 3 ]
 p31 := p15.1; [ 108 ]

CYCLE [ 4 ]
 p31 := p15.1; [ 116 ]

CYCLE [ 5 ]
 p31 := p15.2; [ 132 ]


CYCLE [ 6 ]
 p31 := p15.2; [ r32 ]

CYCLE [ 7 ]
 p31 := p15.4; [ r64 ]


CYCLE [ 8 ]
 p31 := p15.1; [ s08 ]

CYCLE [ 9 ]
 p31 := p15.1; [ s16 ]

CYCLE [ 10 ]
 p31 := p15.2; [ s32 ]


CYCLE [ 11 ]
 p31 := p15.1; [ u08 ]

CYCLE [ 12 ]
 p31 := p15.1; [ u16 ]

CYCLE [ 13 ]
 p31 := p15.2; [ u32 ]
```

Figure C-6. Target Processor C Example - PROCESS.TXT

```
p00 input.bl input.txt <null>
p31 output.bl <null> output.txt
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
```

Figure C-7. Target Processor C Example - INPUT.TXT

```
# c32
complex_32bit
1
1.2 -1.2
# c64
complex_64bit
1
12.12 -12.12
# l08
logical_08bit
1
true
# l16
logical_16bit
1
false
# l32
logical_32bit
1
true
# r32
real_32bit
1
1.2
# r64
real_64bit
1
```

```
12.12
# s08
signed_08bit
1
-12
# s16
signed_16bit
1
-1234
# s32
signed_32bit
1
-12345678
# u08
unsigned_08bit
1
12
# u16
unsigned_16bit
1
1234
# u32
unsigned_32bit
1
12345678
```

Figure C-8. Target Processor C Example - OUTPUT.TXT

```
c32= ( 1.2, -1.2 )
c64= ( 12.12, -12.12 )
l08= true
l16= false
l32= true
r32= 1.2
r64= 12.12
s08= -12
s16= -1234
s32= -12345678
u08= 0x12
u16= 0x1234
u32= 0x12345678
```

## 6.4 Appendix D

## Target Processor FORTRAN I/O

The FORTRAN variable types supported by the I/O routines are described in Table D-1

**Table D-1 Supported I/O variable types for FORTRAN.**

| data_type | alternate data_type | message_type | network |
|---|---|---|---|
| character | | CHARACTER_08BIT | p1:=p0.1 |
| complex*8 | | COMPLEX_32BIT | p1:=p0.4 |
| complex*16 | | COMPLEX_64BIT | p1:=p0.8 |
| logical*1 | | LOGICAL_08BIT | p1:=p0.1 |
| logical*2 | | LOGICAL_16BIT | p1:=p0.1 |
| logical*4 | | LOGICAL_32BIT | p1:=p0.2 |
| real*4 | | REAL_32BIT | p1:=p0.2 |
| real*8 | | REAL_64BIT | p1:=p0.4 |
| integer*1 | | SIGNED_08BIT | p1:=p0.1 |
| integer*2 | | SIGNED_16BIT | p1:=p0.1 |
| integer*4 | | SIGNED_32BIT | p1:=p0.2 |
| integer*1 | | UNSIGNED_08BIT | p1:=p0.1 |
| integer*2 | | UNSIGNED_16BIT | p1:=p0.1 |
| integer*4 | | UNSIGNED_32BIT | p1:=p0.2 |

The Intel iRMXII FORTRAN compiler supports including files at compile time. As such we can include files that have definitions for seperately compiled functions and procedures. These files usually have the extension '.for' or '.inc'.

The procedures used to send and receive data between target processors follow the form:

**call send_message_type(message)** and

**call receive_message_type(message)**

where **message_type** is replaced with an entry from Table D-1 and where message is a scalar variable and is the corresponding data_type or alternate data_type.

The procedures used to send and receive data between the target processors and the host follow the form:

**call output_message(%VAL(message_type),message,%VAL(message_size))** and

**call input_message(message_type,message,message_size)**

where message_type is replaced with an entry from Table D-1, where message is a scalar or array variable and is the corresponding data_type or alternate data_type, and where message_size is the number of datums in message.

Figure D-1. Target Processor FORTRAN Example - MAKEFILE

```
forflags = code large optimize(3)

default:    input.bl output.bl crossbar.bl sequencer.bl

input.bl:  input.obj
    submit :PFP:csd/forbldl( input, input.obj )

input.obj: input.for
    ftn286 input.for $(forflags)

output.bl: output.obj
    submit :PFP:csd/forbldl( output, output.obj )

output.obj:output.for
    ftn286 output.for $(forflags)

crossbar.bl sequencer.bl: network.txt
    submit :PFP:csd/xbc( network.txt )

clean:
    delete *.lst,*.obj,*.mp?,*.bl

run:        input.bl output.bl crossbar.bl sequencer.bl
    reset
    download process.txt
    start process.txt
    ioserve process.txt 2
```

Figure D-2. Target Processor FORTRAN Example - TARGET.FOR

```
$NOLIST

    INTEGER*2 CHARACTER_08BIT
    PARAMETER ( CHARACTER_08BIT = 0 )

    INTEGER*2 COMPLEX_32BIT
    PARAMETER ( COMPLEX_32BIT = 1 )
    INTEGER*2 COMPLEX_64BIT
    PARAMETER ( COMPLEX_64BIT = 2 )

    INTEGER*2 LOGICAL_08BIT
    PARAMETER ( LOGICAL_08BIT = 3 )
    INTEGER*2 LOGICAL_16BIT
    PARAMETER ( LOGICAL_16BIT = 4 )
    INTEGER*2 LOGICAL_32BIT
    PARAMETER ( LOGICAL_32BIT = 5 )

    INTEGER*2 REAL_32BIT
    PARAMETER ( REAL_32BIT = 6 )
    INTEGER*2 REAL_64BIT
    PARAMETER ( REAL_64BIT = 7 )

    INTEGER*2 SIGNED_08BIT
    PARAMETER ( SIGNED_08BIT = 8 )
    INTEGER*2 SIGNED_16BIT
    PARAMETER ( SIGNED_16BIT = 9 )
    INTEGER*2 SIGNED_32BIT
    PARAMETER ( SIGNED_32BIT = 10 )

    INTEGER*2 UNSIGNED_08BIT
    PARAMETER ( UNSIGNED_08BIT = 11 )
    INTEGER*2 UNSIGNED_16BIT
    PARAMETER ( UNSIGNED_16BIT = 12 )
    INTEGER*2 UNSIGNED_32BIT
    PARAMETER ( UNSIGNED_32BIT = 13 )

    LOGICAL*1 INPUT_READY
    LOGICAL*1 OUTPUT_READY

$LIST
```

Figure D-3. Target Processor FORTRAN Example - INPUT.FOR

```
    program main

$include(':PFP:include/target.for')

    INTEGER*2 message_type
    INTEGER*2 message_size

    COMPLEX*8 c32
    COMPLEX*16 c64

    LOGICAL*1 l08
    LOGICAL*2 l16
    LOGICAL*4 l32

    REAL*4 r32
    REAL*8 r64

    INTEGER*1 s08
    INTEGER*2 s16
    INTEGER*4 s32

    INTEGER*1 u08
    INTEGER*2 u16
    INTEGER*4 u32

    call input_message( message_type, c32, message_size )
    call input_message( message_type, c64, message_size )

    call input_message( message_type, l08, message_size )
    call input_message( message_type, l16, message_size )
    call input_message( message_type, l32, message_size )

    call input_message( message_type, r32, message_size )
    call input_message( message_type, r64, message_size )

    call input_message( message_type, s08, message_size )
    call input_message( message_type, s16, message_size )
    call input_message( message_type, s32, message_size )

    call input_message( message_type, u08, message_size )
    call input_message( message_type, u16, message_size )+
    call input_message( message_type, u32, message_size )

    call send_COMPLEX_32BIT( c32 )
    call send_COMPLEX_64BIT( c64 )

    call send_LOGICAL_08BIT( l08 )
    call send_LOGICAL_16BIT( l16 )
    call send_LOGICAL_32BIT( l32 )

    call send_REAL_32BIT( r32 )
    call send_REAL_64BIT( r64 )

    call send_SIGNED_08BIT( s08 )
    call send_SIGNED_16BIT( s16 )
    call send_SIGNED_32BIT( s32 )

    call send_UNSIGNED_08BIT( u08 )
    call send_UNSIGNED_16BIT( u16 )
    call send_UNSIGNED_32BIT( u32 )

    end
```

Figure D-4. Target Processor FORTRAN Example - OUTPUT.FOR

```
    program main

$include(':PFP:include/target.for')

    COMPLEX*8 c32
    COMPLEX*16 c64
```

```
LOGICAL*1 l08
LOGICAL*2 l16
LOGICAL*4 l32

REAL*4 r32
REAL*8 r64

INTEGER*1 s08
INTEGER*2 s16
INTEGER*4 s32

INTEGER*1 u08
INTEGER*2 u16
INTEGER*4 u32

call receive_COMPLEX_32BIT( c32 )
call receive_COMPLEX_64BIT( c64 )

call receive_LOGICAL_08BIT( l08 )
call receive_LOGICAL_16BIT( l16 )
call receive_LOGICAL_32BIT( l32 )

call receive_REAL_32BIT( r32 )
call receive_REAL_64BIT( r64 )

call receive_SIGNED_08BIT( s08 )
call receive_SIGNED_16BIT( s16 )
call receive_SIGNED_32BIT( s32 )

call receive_UNSIGNED_08BIT( u08 )
call receive_UNSIGNED_16BIT( u16 )
call receive_UNSIGNED_32BIT( u32 )

call output_message( %VAL(CHARACTER_08BIT), 'c32= ' )
call output_message( %VAL(COMPLEX_32BIT), c32, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 'c64= ' )
call output_message( %VAL(COMPLEX_64BIT), c64, %VAL(1) )
call output_nl

call output_message( %VAL(CHARACTER_08BIT), 'l08= ' )
call output_message( %VAL(LOGICAL_08BIT), l08, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 'l16= ' )
call output_message( %VAL(LOGICAL_16BIT), l16, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 'l32= ' )
call output_message( %VAL(LOGICAL_32BIT), l32, %VAL(1) )
call output_nl

call output_message( %VAL(CHARACTER_08BIT), 'r32= ' )
call output_message( %VAL(REAL_32BIT), r32, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 'r64= ' )
call output_message( %VAL(REAL_64BIT), r64, %VAL(1) )
call output_nl

call output_message( %VAL(CHARACTER_08BIT), 's08= ' )
call output_message( %VAL(SIGNED_08BIT), s08, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 's16= ' )
call output_message( %VAL(SIGNED_16BIT), s16, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 's32= ' )
call output_message( %VAL(SIGNED_32BIT), s32, %VAL(1) )
call output_nl

call output_message( %VAL(CHARACTER_08BIT), 'u08= ' )
call output_message( %VAL(UNSIGNED_08BIT), u08, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 'u16= ' )
call output_message( %VAL(UNSIGNED_16BIT), u16, %VAL(1) )
call output_nl
call output_message( %VAL(CHARACTER_08BIT), 'u32= ' )
call output_message( %VAL(UNSIGNED_32BIT), u32, %VAL(1) )
call output_nl

end
```

Figure D-5. Target Processor FORTRAN Example - NETWORK.TXT

```
LOOP;


CYCLE [ 1 ]
 p31 := p15.4; [ c32 ]

CYCLE [ 2 ]
 p31 := p15.8; [ c64 ]


CYCLE [ 3 ]
 p31 := p15.1; [ 108 ]

CYCLE [ 4 ]
 p31 := p15.1; [ 116 ]

CYCLE [ 5 ]
 p31 := p15.2; [ 132 ]


CYCLE [ 6 ]
 p31 := p15.2; [ r32 ]

CYCLE [ 7 ]
 p31 := p15.4; [ r64 ]


CYCLE [ 8 ]
 p31 := p15.1; [ s08 ]

CYCLE [ 9 ]
 p31 := p15.1; [ s16 ]

CYCLE [ 10 ]
 p31 := p15.2; [ s32 ]


CYCLE [ 11 ]
 p31 := p15.1; [ u08 ]

CYCLE [ 12 ]
 p31 := p15.1; [ u16 ]

CYCLE [ 13 ]
 p31 := p15.2; [ u32 ]
```

Figure D-6. Target Processor FORTRAN Example - PROCESS.TXT

```
p00 input.bl input.txt <null>
p31 output.bl <null> output.txt
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
```

Figure D-7. Target Processor FORTRAN Example - INPUT.TXT

```
# c32
complex_32bit
1
1.2 -1.2
# c64
complex_64bit
1
12.12 -12.12
# 108
logical_08bit
1
true
# 116
logical_16bit
```

```
1
false
# l32
logical_32bit
1
true
# r32
real_32bit
1
1.2
# r64
real_64bit
1
12.12
# s08
signed_08bit
1
-12
# s16
signed_16bit
1
-1234
# s32
signed_32bit
1
-12345678
# u08
unsigned_08bit
1
12
# u16
unsigned_16bit
1
1234
# u32
unsigned_32bit
1
12345678
```

Figure D-8. Target Processor FORTRAN Example - OUTPUT.TXT

```
c32= ( 1.2, -1.2 )
c64= ( 12.12, -12.12 )
l08= true
l16= false
l32= true
r32= 1.2
r64= 12.12
s08= -12
s16= -1234
s32= -12345678
u08= 0x12
u16= 0x1234
u32= 0x12345678
```

## 6.3 Appendix E

## Target Processor PASCAL I/O

The PASCAL variable types supported by the I/O routines are described in Table E-1.

**Table E-1 Supported I/O variable types for PASCAL**

| data_type | alternate data_type | message_type | network |
|---|---|---|---|
| char | CHARACTER_08BIT_type | CHARACTER_08BIT | p1:=p0.1 |
| record<br>r: real;<br>i: real;<br>end | COMPLEX_32BIT_type | COMPLEX_32BIT | p1:=p0.4 |
| record<br>r: longreal;<br>i: longreal;<br>end | COMPLEX_64BIT_type | COMPLEX_64BIT | p1:=p0.8 |
| boolean | LOGICAL_08BIT_type | LOGICAL_08BIT | p1:=p0.1 |
| integer | LOGICAL_16BIT_type | LOGICAL_16BIT | p1:=p0.1 |
| longint | LOGICAL_32BIT_type | LOGICAL_32BIT | p1:=p0.2 |
| real | REAL_32BIT_type | REAL_32BIT | p1:=p0.2 |
| longreal | REAL_64BIT_type | REAL_64BIT | p1:=p0.4 |
| 0..255 | SIGNED_08BIT_type | SIGNED_08BIT | p1:=p0.1 |
| integer | SIGNED_16BIT_type | SIGNED_16BIT | p1:=p0.1 |
| longint | SIGNED_32BIT_type | SIGNED_32BIT | p1:=p0.2 |
| 0..255 | UNSIGNED_08BIT_type | UNSIGNED_08BIT | p1:=p0.1 |
| word | UNSIGNED_16BIT_type | UNSIGNED_16BIT | p1:=p0.1 |
| longint | UNSIGNED_32BIT_type | UNSIGNED_32BIT | p1:=p0.2 |

The Intel iRMXII PASCAL compiler supports including files at compile time. As such we can include files that have definitions for seperately compiled functions and procedures. These files usually have the extension '.pas' or '.inc'.

The procedures used to send and receive data between target processors follow the form:

**send_message_type(message);** and

**receive_message_type(message);**

where **message_type** is replaced with an entry from Table E-1 and where message is a scalar variable and is the corresponding data_type or alternate data_type.

The procedures used to send and receive data between the target processors and the host follow the form:

**output_message(message_type,message,message_size);** and

**input_message(message_type,message,message_size);**

where message_type is replaced with an entry from Table E-1, where message is a scalar or array variable and is the corresponding data_type or alternate data_type, and where message_size is the number of datums in message.

Figure E-1. Target Processor PASCAL Example - MAKEFILE

```
pasflags = code large optimize(1) symbolspace(64)

default:   input.bl output.bl crossbar.bl sequencer.bl

input.bl:  input.obj
    submit :PFP:csd/pasbldl( input, input.obj )

input.obj: input.pas
    pas286 input.pas $(pasflags)

output.bl: output.obj
    submit :PFP:csd/pasbldl( output, output.obj )

output.obj:output.pas
    pas286 output.pas $(pasflags)

crossbar.bl sequencer.bl: network.txt
    submit :PFP:csd/xbc( network.txt )

clean:
    delete *.lst,*.obj,*.mp?,*.bl

run:       input.bl output.bl crossbar.bl sequencer.bl
    reset
    download process.txt
    start process.txt
    ioserve process.txt 2
```

Figure E-2. Target Processor PASCAL Example - TARGET.PAS

```
$NOLIST

public target;

const
    CHARACTER_08BIT = 0;
    COMPLEX_32BIT = 1;
    COMPLEX_64BIT = 2;
    LOGICAL_08BIT = 3;
    LOGICAL_16BIT = 4;
    LOGICAL_32BIT = 5;
    REAL_32BIT = 6;
    REAL_64BIT = 7;
    SIGNED_08BIT = 8;
    SIGNED_16BIT = 9;
    SIGNED_32BIT = 10;
    UNSIGNED_08BIT = 11;
    UNSIGNED_16BIT = 12;
    UNSIGNED_32BIT = 13;

type
    CHARACTER_08BIT_type = char;
    COMPLEX_32BIT_type = record r: real; i: real; end;
    COMPLEX_64BIT_type = record r: longreal; i: longreal; end;
    LOGICAL_08BIT_type = boolean;
    LOGICAL_16BIT_type = integer;
    LOGICAL_32BIT_type = longint;
    REAL_32BIT_type = real;
    REAL_64BIT_type = longreal;
    SIGNED_08BIT_type = 0..255;
    SIGNED_16BIT_type = integer;
    SIGNED_32BIT_type = longint;
    UNSIGNED_08BIT_type = 0..255;
    UNSIGNED_16BIT_type = word;
```

```
    UNSIGNED_32BIT_type = longint;

procedure output_nl;

procedure input_buffer( var buffer: bytes; buffer_size: integer );
procedure input_message( var message_type: integer;
    var message: bytes; var message_size: integer );
function input_ready: boolean;

procedure led( state: integer );

procedure output_buffer( var buffer: bytes; buffer_size: integer );
procedure output_message( message_type: integer;
    var message: bytes; message_size: integer );
function output_ready: boolean;

procedure receive_buffer( var buffer: bytes; buffer_size: integer );
procedure receive_COMPLEX_32BIT( var buffer: COMPLEX_32BIT_type );
procedure receive_COMPLEX_64BIT( var buffer: COMPLEX_64BIT_type );
procedure receive_LOGICAL_08BIT( var buffer: LOGICAL_08BIT_type );
procedure receive_LOGICAL_16BIT( var buffer: LOGICAL_16BIT_type );
procedure receive_LOGICAL_32BIT( var buffer: LOGICAL_32BIT_type );
procedure receive_REAL_32BIT( var buffer: REAL_32BIT_type );
procedure receive_REAL_64BIT( var buffer: REAL_64BIT_type );
procedure receive_SIGNED_08BIT( var buffer: SIGNED_08BIT_type );
procedure receive_SIGNED_16BIT( var buffer: SIGNED_16BIT_type );
procedure receive_SIGNED_32BIT( var buffer: SIGNED_32BIT_type );
procedure receive_UNSIGNED_08BIT( var buffer: UNSIGNED_08BIT_type );
procedure receive_UNSIGNED_16BIT( var buffer: UNSIGNED_16BIT_type );
procedure receive_UNSIGNED_32BIT( var buffer: UNSIGNED_32BIT_type );

procedure send_buffer( var buffer: bytes; buffer_size: integer );
procedure send_COMPLEX_32BIT( var buffer: COMPLEX_32BIT_type );
procedure send_COMPLEX_64BIT( var buffer: COMPLEX_64BIT_type );
procedure send_LOGICAL_08BIT( var buffer: LOGICAL_08BIT_type );
procedure send_LOGICAL_16BIT( var buffer: LOGICAL_16BIT_type );
procedure send_LOGICAL_32BIT( var buffer: LOGICAL_32BIT_type );
procedure send_REAL_32BIT( var buffer: REAL_32BIT_type );
procedure send_REAL_64BIT( var buffer: REAL_64BIT_type );
procedure send_SIGNED_08BIT( var buffer: SIGNED_08BIT_type );
procedure send_SIGNED_16BIT( var buffer: SIGNED_16BIT_type );
procedure send_SIGNED_32BIT( var buffer: SIGNED_32BIT_type );
procedure send_UNSIGNED_08BIT( var buffer: UNSIGNED_08BIT_type );
procedure send_UNSIGNED_16BIT( var buffer: UNSIGNED_16BIT_type );
procedure send_UNSIGNED_32BIT( var buffer: UNSIGNED_32BIT_type );

$LIST
```

Figure E-3. Target Processor PASCAL Example - INPUT.PAS

```
module main;

$include(':PFP:include/target.pas')

program main;

var
    message_type: SIGNED_16BIT_type;
    message_size: SIGNED_16BIT_type;

    c32: COMPLEX_32BIT_type;
    c64: COMPLEX_64BIT_type;
    l08: LOGICAL_08BIT_type;
    l16: LOGICAL_16BIT_type;
    l32: LOGICAL_32BIT_type;

    r32: REAL_32BIT_type;
    r64: REAL_64BIT_type;

    s08: SIGNED_08BIT_type;
    s16: SIGNED_16BIT_type;
    s32: SIGNED_32BIT_type;

    u08: UNSIGNED_08BIT_type;
    u16: UNSIGNED_16BIT_type;
```

```
    u32: UNSIGNED_32BIT_type;

begin
    input_message( message_type, c32, message_size );
    input_message( message_type, c64, message_size );

    input_message( message_type, l08, message_size );
    input_message( message_type, l16, message_size );
    input_message( message_type, l32, message_size );

    input_message( message_type, r32, message_size );
    input_message( message_type, r64, message_size );

    input_message( message_type, s08, message_size );
    input_message( message_type, s16, message_size );
    input_message( message_type, s32, message_size );

    input_message( message_type, u08, message_size );
    input_message( message_type, u16, message_size );
    input_message( message_type, u32, message_size );

    send_COMPLEX_32BIT( c32 );
    send_COMPLEX_64BIT( c64 );

    send_LOGICAL_08BIT( l08 );
    send_LOGICAL_16BIT( l16 );
    send_LOGICAL_32BIT( l32 );

    send_REAL_32BIT( r32 );
    send_REAL_64BIT( r64 );

    send_SIGNED_08BIT( s08 );
    send_SIGNED_16BIT( s16 );
    send_SIGNED_32BIT( s32 );

    send_UNSIGNED_08BIT( u08 );
    send_UNSIGNED_16BIT( u16 );
    send_UNSIGNED_32BIT( u32 );

end { main }
```

Figure E-4. Target Processor PASCAL Example - OUTPUT.PAS

```
module main;

$include(':PFP:include/target.pas')

program main;

var
    c32: COMPLEX_32BIT_type;
    c64: COMPLEX_64BIT_type;

    l08: LOGICAL_08BIT_type;
    l16: LOGICAL_16BIT_type;
    l32: LOGICAL_32BIT_type;

    r32: REAL_32BIT_type;
    r64: REAL_64BIT_type;

    s08: SIGNED_08BIT_type;
    s16: SIGNED_16BIT_type;
    s32: SIGNED_32BIT_type;

    u08: UNSIGNED_08BIT_type;
    u16: UNSIGNED_16BIT_type;
    u32: UNSIGNED_32BIT_type;

begin
    receive_COMPLEX_32BIT( c32 );
    receive_COMPLEX_64BIT( c64 );

    receive_LOGICAL_08BIT( l08 );
    receive_LOGICAL_16BIT( l16 );
    receive_LOGICAL_32BIT( l32 );
```

```
    receive_REAL_32BIT( r32 );
    receive_REAL_64BIT( r64 );

    receive_SIGNED_08BIT( s08 );
    receive_SIGNED_16BIT( s16 );
    receive_SIGNED_32BIT( s32 );

    receive_UNSIGNED_08BIT( u08 );
    receive_UNSIGNED_16BIT( u16 );
    receive_UNSIGNED_32BIT( u32 );

    output_message( CHARACTER_08BIT, 'c32= ', 5 );
    output_message( COMPLEX_32BIT, c32, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 'c64= ', 5 );
    output_message( COMPLEX_64BIT, c64, 1 );
    output_nl;

    output_message( CHARACTER_08BIT, 'l08= ', 5 );
    output_message( LOGICAL_08BIT, l08, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 'l16= ', 5 );
    output_message( LOGICAL_16BIT, l16, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 'l32= ', 5 );
    output_message( LOGICAL_32BIT, l32, 1 );
    output_nl;

    output_message( CHARACTER_08BIT, 'r32= ', 5 );
    output_message( REAL_32BIT, r32, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 'r64= ', 5 );
    output_message( REAL_64BIT, r64, 1 );
    output_nl;

    output_message( CHARACTER_08BIT, 's08= ', 5 );
    output_message( SIGNED_08BIT, s08, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 's16= ', 5 );
    output_message( SIGNED_16BIT, s16, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 's32= ', 5 );
    output_message( SIGNED_32BIT, s32, 1 );
    output_nl;

    output_message( CHARACTER_08BIT, 'u08= ', 5 );
    output_message( UNSIGNED_08BIT, u08, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 'u16= ', 5 );
    output_message( UNSIGNED_16BIT, u16, 1 );
    output_nl;
    output_message( CHARACTER_08BIT, 'u32= ', 5 );
    output_message( UNSIGNED_32BIT, u32, 1 );
    output_nl;

end { main }
```

Figure E-5. Target Processor PASCAL Example - NETWORK.TXT

```
LOOP;


CYCLE [ 1 ]
 p31 := p15.4; [ c32 ]

CYCLE [ 2 ]
 p31 := p15.8; [ c64 ]


CYCLE [ 3 ]
 p31 := p15.1; [ 108 ]

CYCLE [ 4 ]
 p31 := p15.1; [ 116 ]
```

```
CYCLE [ 5 ]
 p31 := p15.2; [ l32 ]


CYCLE [ 6 ]
 p31 := p15.2; [ r32 ]

CYCLE [ 7 ]
 p31 := p15.4; [ r64 ]


CYCLE [ 8 ]
 p31 := p15.1; [ s08 ]

CYCLE [ 9 ]
 p31 := p15.1; [ s16 ]

CYCLE [ 10 ]
 p31 := p15.2; [ s32 ]


CYCLE [ 11 ]
 p31 := p15.1; [ u08 ]

CYCLE [ 12 ]
 p31 := p15.1; [ u16 ]

CYCLE [ 13 ]
 p31 := p15.2; [ u32 ]
```

Figure E-6. Target Processor PASCAL Example - PROCESS.TXT

```
p00 input.bl input.txt <null>
p31 output.bl <null> output.txt
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
```

Figure E-7. Target Processor PASCAL Example - INPUT.TXT

```
# c32
complex_32bit
1
1.2 -1.2
# c64
complex_64bit
1
12.12 -12.12
# l08
logical_08bit
1
true
# l16
logical_16bit
1
false
# l32
logical_32bit
1
true
# r32
real_32bit
1
1.2
# r64
real_64bit
1
12.12
# s08
signed_08bit
1
-12
```

```
# s16
signed_16bit
1
-1234
# s32
signed_32bit
1
-12345678
# u08
unsigned_08bit
1
12
# u16
unsigned_16bit
1
1234
# u32
unsigned_32bit
1
12345678
```

Figure E-8. Target Processor PASCAL Example - OUTPUT.TXT

```
c32= ( 1.2, -1.2 )
c64= ( 12.12, -12.12 )
l08= true
l16= false
l32= true
r32= 1.2
r64= 12.12
s08= -12
s16= -1234
s32= -12345678
u08= 0x12
u16= 0x1234
u32= 0x12345678
```

## 6.5 Appendix F

# Target Processor PLM I/O

The PLM variable types supported by the I/O routines are described in Table F-1.

**Table F-1 Supported I/O variable types for PLM.**

| data_type | alternate data_type | message_type | network |
|---|---|---|---|
| byte | CHARACTER_08BIT_type | CHARACTER_08BIT | p1:=p0.1 |
| structure<br>(<br>r real,<br>i real<br>) | COMPLEX_32BIT_type | COMPLEX_32BIT | p1:=p0.4 |
| | | COMPLEX_64BIT | |
| byte | LOGICAL_08BIT_type | LOGICAL_08BIT | p1:=p0.1 |
| word | LOGICAL_16BIT_type | LOGICAL_16BIT | p1:=p0.1 |
| dword | LOGICAL_32BIT_type | LOGICAL_32BIT | p1:=p0.2 |
| real | REAL_32BIT_type | REAL_32BIT | p1:=p0.2 |
| | | REAL_64BIT | |
| byte | SIGNED_08BIT_type | SIGNED_08BIT | p1:=p0.1 |
| integer | SIGNED_16BIT_type | SIGNED_16BIT | p1:=p0.1 |
| dword | SIGNED_32BIT_type | SIGNED_32BIT | p1:=p0.2 |
| byte | UNSIGNED_08BIT_type | UNSIGNED_08BIT | p1:=p0.1 |
| word | UNSIGNED_16BIT_type | UNSIGNED_16BIT | p1:=p0.1 |
| dword | UNSIGNED_32BIT_type | UNSIGNED_32BIT | p1:=p0.2 |

The Intel iRMXII PLM compiler supports including files at compile time. As such we can include files that have definitions for seperately compiled functions and procedures. These files usually have the extension '.plm' or '.inc'.

The procedures used to send and receive data between target processors follow the form:

**call send_message_type(@message);** and

**call receive_message_type(@message);**

where **message_type** is replaced with an entry from Table F-1 and where message is a scalar variable and is the corresponding data_type or alternate data_type.

The procedures used to send and receive data between the target processors and the host follow the form:

**call output_message(message_type,@message,message_size);** and

**call input_message(@message_type,@message,@message_size);**

where message_type is replaced with an entry from Table F-1, where message is a scalar or array variable and is the corresponding data_type or alternate data_type, and where message_size is the number of datums in message.

Figure F-1. Target Processor PLM Example - MAKEFILE

```
plmflags = code large optimize(3)

default:   input.bl output.bl crossbar.bl sequencer.bl

input.bl:  input.obj
    submit :PFP:csd/plmbldl( input, input.obj )

input.obj: input.plm
    plm286 input.plm $(plmflags)

output.bl: output.obj
    submit :PFP:csd/plmbldl( output, output.obj )

output.obj:output.plm
    plm286 output.plm $(plmflags)

crossbar.bl sequencer.bl: network.txt
    submit :PFP:csd/xbc( network.txt )

clean:
    delete *.lst,*.obj,*.mp?,*.bl

run:       input.bl output.bl crossbar.bl sequencer.bl
    reset
    download process.txt
    start process.txt
    ioserve process.txt 2
```

Figure F-2. Target Processor PLM Example - TARGET.PLM

```
$NOLIST

declare CHARACTER_08BIT literally '0';
declare COMPLEX_32BIT literally '1';
/* declare COMPLEX_64BIT literally '2'; */
declare LOGICAL_08BIT literally '3';
declare LOGICAL_16BIT literally '4';
declare LOGICAL_32BIT literally '5';
declare REAL_32BIT literally '6';
/* declare REAL_64BIT literally '7'; */
declare SIGNED_08BIT literally '8';
declare SIGNED_16BIT literally '9';
declare SIGNED_32BIT literally '10';
declare UNSIGNED_08BIT literally '11';
declare UNSIGNED_16BIT literally '12';
declare UNSIGNED_32BIT literally '13';

declare CHARACTER_08BIT_type literally 'byte';
declare COMPLEX_32BIT_type literally 'structure ( r real, i real )';
declare LOGICAL_08BIT_type literally 'byte';
declare LOGICAL_16BIT_type literally 'word';
declare LOGICAL_32BIT_type literally 'dword';
declare REAL_32BIT_type literally 'real';
declare SIGNED_08BIT_type literally 'byte';
declare SIGNED_16BIT_type literally 'integer';
declare SIGNED_32BIT_type literally 'dword';
declare UNSIGNED_08BIT_type literally 'byte';
declare UNSIGNED_16BIT_type literally 'word';
declare UNSIGNED_32BIT_type literally 'dword';

output_nl: procedure external;
end output_nl;

input_buffer: procedure( buffer, buffer_size ) external;
```

```
    declare buffer pointer;
    declare buffer_size integer;
end input_buffer;

input_message: procedure( message_type, message, message_size ) external;
    declare message_type pointer;
    declare message pointer;
    declare message_size pointer;
end input_message;

input_ready: procedure byte external;
end input_ready;

led: procedure( state ) external;
    declare state integer;
end led;

output_buffer: procedure( buffer, buffer_size ) external;
    declare buffer pointer;
    declare buffer_size integer;
end output_buffer;

output_message: procedure( message_type, message, message_size ) external;
    declare message_type integer;
    declare message pointer;
    declare message_size integer;
end output_message;

output_ready: procedure byte external;
end output_ready;

receive_buffer: procedure( buffer, buffer_size ) external;
    declare buffer pointer;
    declare buffer_size integer;
end receive_buffer;

receive_CHARACTER_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_CHARACTER_08BIT;

receive_COMPLEX_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_COMPLEX_32BIT;

receive_LOGICAL_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_LOGICAL_08BIT;

receive_LOGICAL_16BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_LOGICAL_16BIT;

receive_LOGICAL_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_LOGICAL_32BIT;

receive_REAL_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_REAL_32BIT;

receive_SIGNED_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_SIGNED_08BIT;

receive_SIGNED_16BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_SIGNED_16BIT;

receive_SIGNED_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_SIGNED_32BIT;

receive_UNSIGNED_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_UNSIGNED_08BIT;

receive_UNSIGNED_16BIT: procedure( buffer ) external;
    declare buffer pointer;
```

```
end receive_UNSIGNED_16BIT;

receive_UNSIGNED_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end receive_UNSIGNED_32BIT;

send_buffer: procedure( buffer, buffer_size ) external;
    declare buffer pointer;
    declare buffer_size integer;
end send_buffer;

send_CHARACTER_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_CHARACTER_08BIT;

send_COMPLEX_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_COMPLEX_32BIT;

send_LOGICAL_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_LOGICAL_08BIT;

send_LOGICAL_16BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_LOGICAL_16BIT;

send_LOGICAL_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_LOGICAL_32BIT;

send_REAL_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_REAL_32BIT;

send_SIGNED_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_SIGNED_08BIT;

send_SIGNED_16BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_SIGNED_16BIT;

send_SIGNED_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_SIGNED_32BIT;

send_UNSIGNED_08BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_UNSIGNED_08BIT;

send_UNSIGNED_16BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_UNSIGNED_16BIT;

send_UNSIGNED_32BIT: procedure( buffer ) external;
    declare buffer pointer;
end send_UNSIGNED_32BIT;

declare FALSE literally '0';
declare TRUE literally '1';

$LIST
```

Figure F-3. Target Processor PLM Example - INPUT.PLM

```
main: do;

$include(':PFP:include/target.plm')

    declare message_type SIGNED_16BIT_type;
    declare message_size SIGNED_16BIT_type;

    declare c32 COMPLEX_32BIT_type;
```

```
    declare l08 LOGICAL_08BIT_type;
    declare l16 LOGICAL_16BIT_type;
    declare l32 LOGICAL_32BIT_type;

    declare r32 REAL_32BIT_type;

    declare s08 SIGNED_08BIT_type;
    declare s16 SIGNED_16BIT_type;
    declare s32 SIGNED_32BIT_type;

    declare u08 UNSIGNED_08BIT_type;
    declare u16 UNSIGNED_16BIT_type;
    declare u32 UNSIGNED_32BIT_type;

    call input_message( @message_type, @c32, @message_size );

    call input_message( @message_type, @l08, @message_size );
    call input_message( @message_type, @l16, @message_size );
    call input_message( @message_type, @l32, @message_size );

    call input_message( @message_type, @r32, @message_size );

    call input_message( @message_type, @s08, @message_size );
    call input_message( @message_type, @s16, @message_size );
    call input_message( @message_type, @s32, @message_size );

    call input_message( @message_type, @u08, @message_size );
    call input_message( @message_type, @u16, @message_size );
    call input_message( @message_type, @u32, @message_size );

    call send_COMPLEX_32BIT( @c32 );

    call send_LOGICAL_08BIT( @l08 );
    call send_LOGICAL_16BIT( @l16 );
    call send_LOGICAL_32BIT( @l32 );

    call send_REAL_32BIT( @r32 );

    call send_SIGNED_08BIT( @s08 );
    call send_SIGNED_16BIT( @s16 );
    call send_SIGNED_32BIT( @s32 );

    call send_UNSIGNED_08BIT( @u08 );
    call send_UNSIGNED_16BIT( @u16 );
    call send_UNSIGNED_32BIT( @u32 );

    halt;
end main;
```

Figure F-4. Target Processor PLM Example - OUTPUT.PLM

```
main: do;

$include(':PFP:include/target.plm')

    declare c32 COMPLEX_32BIT_type;

    declare l08 LOGICAL_08BIT_type;
    declare l16 LOGICAL_16BIT_type;
    declare l32 LOGICAL_32BIT_type;

    declare r32 REAL_32BIT_type;

    declare s08 SIGNED_08BIT_type;
    declare s16 SIGNED_16BIT_type;
    declare s32 SIGNED_32BIT_type;

    declare u08 UNSIGNED_08BIT_type;
    declare u16 UNSIGNED_16BIT_type;
    declare u32 UNSIGNED_32BIT_type;

    call receive_COMPLEX_32BIT( @c32 );

    call receive_LOGICAL_08BIT( @l08 );
    call receive_LOGICAL_16BIT( @l16 );
```

```
   call receive_LOGICAL_32BIT( @l32 );

   call receive_REAL_32BIT( @r32 );

   call receive_SIGNED_08BIT( @s08 );
   call receive_SIGNED_16BIT( @s16 );
   call receive_SIGNED_32BIT( @s32 );

   call receive_UNSIGNED_08BIT( @u08 );
   call receive_UNSIGNED_16BIT( @u16 );
   call receive_UNSIGNED_32BIT( @u32 );

   call output_message( CHARACTER_08BIT, @('c32= '), 5 );
   call output_message( COMPLEX_32BIT, @c32, 1 );
   call output_nl;

   call output_message( CHARACTER_08BIT, @('l08= '), 5 );
   call output_message( LOGICAL_08BIT, @l08, 1 );
   call output_nl;
   call output_message( CHARACTER_08BIT, @('l16= '), 5 );
   call output_message( LOGICAL_16BIT, @l16, 1 );
   call output_nl;
   call output_message( CHARACTER_08BIT, @('l32= '), 5 );
   call output_message( LOGICAL_32BIT, @l32, 1 );
   call output_nl;

   call output_message( CHARACTER_08BIT, @('r32= '), 5 );
   call output_message( REAL_32BIT, @r32, 1 );
   call output_nl;

   call output_message( CHARACTER_08BIT, @('s08= '), 5 );
   call output_message( SIGNED_08BIT, @s08, 1 );
   call output_nl;
   call output_message( CHARACTER_08BIT, @('s16= '), 5 );
   call output_message( SIGNED_16BIT, @s16, 1 );
   call output_nl;
   call output_message( CHARACTER_08BIT, @('s32= '), 5 );
   call output_message( SIGNED_32BIT, @s32, 1 );
   call output_nl;

   call output_message( CHARACTER_08BIT, @('u08= '), 5 );
   call output_message( UNSIGNED_08BIT, @u08, 1 );
   call output_nl;
   call output_message( CHARACTER_08BIT, @('u16= '), 5 );
   call output_message( UNSIGNED_16BIT, @u16, 1 );
   call output_nl;
   call output_message( CHARACTER_08BIT, @('u32= '), 5 );
   call output_message( UNSIGNED_32BIT, @u32, 1 );
   call output_nl;

   halt;
end main;
```

Figure F-5. Target Processor PLM Example - NETWORK.TXT

```
LOOP;


CYCLE [ 1 ]
 p31 := p15.4; [ c32 ]


CYCLE [ 2 ]
 p31 := p15.1; [ 108 ]

CYCLE [ 3 ]
 p31 := p15.1; [ 116 ]

CYCLE [ 4 ]
 p31 := p15.2; [ 132 ]


CYCLE [ 5 ]
 p31 := p15.2; [ r32 ]
```

```
CYCLE [ 6 ]
 p31 := p15.1; [ s08 ]

CYCLE [ 7 ]
 p31 := p15.1; [ s16 ]

CYCLE [ 8 ]
 p31 := p15.2; [ s32 ]


CYCLE [ 9 ]
 p31 := p15.1; [ u08 ]

CYCLE [ 10 ]
 p31 := p15.1; [ u16 ]

CYCLE [ 11 ]
 p31 := p15.2; [ u32 ]
```

Figure F-6. Target Processor PLM Example - PROCESS.TXT

```
p00 input.bl input.txt <null>
p31 output.bl <null> output.txt
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
```

Figure F-7. Target Processor PLM Example - INPUT.TXT

```
# c32
complex_32bit
1
1.2 -1.2
# l08
logical_08bit
1
true
# l16
logical_16bit
1
false
# l32
logical_32bit
1
true
# r32
real_32bit
1
1.2
# s08
signed_08bit
1
-12
# s16
signed_16bit
1
-1234
# s32
signed_32bit
1
-12345678
# u08
unsigned_08bit
1
12
# u16
unsigned_16bit
1
1234
# u32
unsigned_32bit
```

```
1
12345678
```

Figure F-8. Target Processor PLM Example - OUTPUT.TXT

```
c32= ( 1.2, -1.2 )
l08= true
l16= false
l32= true
r32= 1.2
s08= -12
s16= -1234
s32= -12345678
u08= 0x12
u16= 0x1234
u32= 0x12345678
```

## 6.5 Appendix G

# Programming Tools

This appendix contains brief explanations of the programming tools that are routinely used to develop programs for the PFP. The programs: reset, download, start, ioserve, and make are discussed.

**RESET: Hardware Reset**

The reset utility is the software implementation of a hardware reset. Reset writes to an address which is interpreted as a reset to the PFP (the crossbar/sequencer and all processing elements.)

**DOWNLOAD: Software Downloader**

The download utility uses an input file, process.txt, to download the appropriate elements on the PFP. Download uses the first two fields of each line in the input file to determine which element to download and with which file to download to the element.

The following is an example input file used with download, start, and ioserve:

```
p00 input.bl input.txt <null>
p31 output.bl <null> output.txt
crossbar crossbar.bl <null> <null>
sequencer sequencer.bl <null> <null>
```

**START: Processing Element Starter**

The start utility uses an input file, process.txt, to start the appropriate elements on the PFP. Start uses only the first field in the input file to determine which elements to start (i.e., begin program execution for the processors and begin microcode execution for the crossbar/sequencer.)

**IOSERVE: Input/Output Host Service Routine**

The ioserve utility is designed to handle any input or output between the host processor and any of the target processors. Ioserve uses an input file, process.txt, to determine whether or not a processing element (a processor) will need any input by examining the third field in each line of the input file and whether or not a processing element will generate any output by examining the fourth field in the input file. If the third field contains a character string other than '<NULL>' the string is assumed to be the name of the input file associated with that processing element. If the fourth field contains a character string other than '<NULL>' the string is assumed to be the name of the output file associated with that processing element. Neither the crossbar nor sequencer support input

and output, therefore, the third and fourth fields in the input file for these elements are '<NULL>'.

If the third field of the input file indicates a processing element requires input (i.e., the third field is a file name), ioserve will open the input file, process the data, and send it to the processing element at the beginning of the execution session. If the fourth field of the input file contains a file name, ioserve writes the output from the processing element to that file. The output is always written to the terminal whether or not an output file is designated. Ioserve in turn scans the data available port for each of the active processing elements by opening the memory window to the processor and checking the appropriate flag. When data is available, ioserve retrieves the data and writes it to the designated output. If no data is available, ioserve closes the window and proceeds onto the next processing element. Ioserve retrieves data from a processing element until the source is exhausted.

Each processing element can have unique input and output files or a combination of shared input and output files. If an input file is shared, each of the processing elements sharing the file should expect the same data as input (i.e., no distinguishing is made for a specific processor in the input file - all data in the input file is sent to the indicated processing elements.) If an output file is shared, the output from all of those processing elements will be intermixed in the output file as it is processed by ioserve. Since this is a parallel environment care will need to be taken when generating output as the order of the output may not be guaranteed.

Single characters, character strings, scalars, and arrays can be sent to the host via the output_message routines. Output is written to the terminal and to the disk in ASCII. Real and integer numbers are written out with six (6) significant digits in either decimal or scientific notation depending on the manitude of the number. The generation of newlines (<CR> and <LF>) is performed by calling the 'output_nl' routine which sends the appropriate characters to the host. This is much like doing a Pascal write and writeln.

The runtime parameter **timeout** has to be used with ioserve. When invoking ioserve, the second parameter on the command line is the input file and the third parameter would be an integer timeout count in seconds. Ioserve scans the active processing elements for output until there has been no output for the specified number of seconds and then ioserve terminates.

**MAKE: Program Maintainer**

Make was originally developed as a project control tool for the UNIX operating system. In UNIX, as in iRMX, most programs are composed of many small source modules that need to be combined together to produce an executable module. Without a utility such as make, it would be necessary for the programmer to keep track of all object modules which might need to be regenerated due to changes in source files. The make program provides an easy

way to automate this process. An iRMX enhanced version of the make program, which is compatible with the UNIX version, is now available to iRMX users.

Make reads commands from a user-defined "makefile" that lists the files to be created, the commands that create them, and the files from which they are created. When you direct make to create a program, it makes sure that each file on which the program depends is up to date, then creates the program by executing the given commands. If a file is not up to date, make updates it before creating the program by executing explicitly given commands or one of the many built-in commands.

Creating a Makefile

A makefile contains one or more lines of text called dependency lines. A dependency line shows how a given file depends on other files and what commands are required to bring a file up to date. A dependency line has the form:

target ... :[dependent...]

    [ command ...]

where 'target' is the name of a file to be updated, 'dependent' is the name of a file on which the target depends, and 'command' is the iRMX command needed to create the target file. Each dependency line must have at least one command associated with it.

You may give more than one target name or dependent name if desired. Each name must be separated from the next by at least one space. The target names must be separated from the dependent names by a colon (:). File names must be spelled as defined by the iRMX system. Note that names are case-sensitive within a makefile.

You may give a sequence of commands on lines following the target by beginning each line with a tab or caret (^) character. Commands must be given exactly as they would appear on an iRMX command line. The at-sign character (@) may be placed in front of a command to prevent make from displaying the command before executing it.

You may add a comment to a makefile by starting the comment with a number sign (#) and ending it with a carriage return. All characters after the number sign are ignored. If a dependency line is too long, you can continue it by typing a backslash (\) immediately followed by a carriage return.

The makefile should be kept in the same directory as the given source files. For convenience, the file name 'makefile' is provided as the default file name used by make if no explicit name is given at invocation. You may use the default name or choose one of your own.

To illustrate dependency lines, consider the following example. A program named test is made by linking three object files, x.obj,y.obj and z.obj. These object files are created by compiling the C language source files x.c,y.c, and z.c. Furthermore, the files x.c and y.c contain the line:

include "defs"

This means test depends on the three object files, the object files depend on the C source files, and two of the source files depend on the include file 'defs'. You can represent these relationships in a make file with the following lines:

```
test: x.obj y.obj z.obj

        BND286 x.obj, y.obj, z.obj, \

        /lib/ic286/clib2c.lib,\

        /lib/ic286/crmx2c.lib, \

        /lib/ic286/cnoflt2c.obj, \

        /lib/ic286/clib2c.lib, \

        /rmx286/ib/rmxifc.lib $(TYPE) nopack \

        segsize(stack(8192)) rc(dm(100,50000)) object($@)

x.obj: x.c defs

        ic286 x.c debug

y.obj: y.c defs

        ic286 y.c debug

z.obj: z.c

        ic286 z.c debug
```

In the first dependency line, test is the target file and x.obj, y.obj, and z.obj are its dependents. The command sequence "BND286 X.obj, y.obj, z.obj, \ /lib/ic286/clib2c, \ /lib/ic286/crmx2c.lib, \ /lib/ic286/cnoflt2c.obj, \ /lib/ic286/clib2clib, \ /rmx286/lib/rmxifc.lib $(TYPE) nopack \ segsize(stack(8192)) rc(dm(100,50000)) object($@)" on the next line tells how to create test if it is out of date. The program is out of date if any one of its dependents has been modified since test was last created.

The second third and fourth dependency lines have the same form, with the x.obj, y.obj, and z.obj files as targets and x.c, y.c, z.c, and 'defs' files as dependents. Each dependency line has one command sequence that defines how to update the given target file.

Invoking Make

Once you have a make file and wish to update and modify one or more target files in the directory, you can invoke make by typing its name and optional arguments. The invocation has the form:

make [option]...[macdef]...[target]...

where 'option' is a program option used to modify program operation, 'macdef' is a macro definition used to give a macro a value or meaning, and 'target' is the name of a file to be updated. 'target' must correspond to one of the target names in the makefile. All arguments are optional. If you give more than one argument, you must separate them with spaces.

You can direct make to update the first target file in the makefile by typing just the program name "make". In this case, make searches for the file Makefile in the current directory. For example, assume that the current makefile contains the dependency lines given in the last section. Then the command **make** compares the modification dates of the test program and each of the object files x.obj, y.obj, and z.obj and recreates test if any changes have been make to any other object files since test was last created. It also compares the modified dates of the object files with those of the four source files, x.c, y.c, z.c, and defs, and recreates the object files if the source files have changed. It does this before recreating test so that the recreated object files can be used to recreate test. If none of the source or object files have been altered since the last time test was made, make announces this fact and stops. No files are changed.

You can direct make to update a given target file by giving the file name of the target. For example,

```
make x.obj
```

causes make to recompile, creating the x.obj files if the x.c or defs files have changed since the object file was last created. Similarly, the command

```
make x.obj z.obj
```

causes make to recompile, creating x.obj and z.obj if the corresponding dependents have been modified. Make processes target names from the command line in a left-to-right order. You can specify the name of the makefile you wish make to use by giving the -f option in the invocation. The option has the form

```
-f filename
```

where filename is the name of the makefile. You must supply a full path name if the file is not in the current directory. For example, the command

```
make -f maketest
```

reads the dependency lines of the makefile named 'maketest' found in the current directory.

If you specify only a target on the command line, and no makefile is present then make will attempt to create the target using only built-in rules. This is nice for small, single module programs.

Using Pseudo-Target Names

It is often useful to include dependency lines that have pseudo-target names, i.e., names for which no files actually exist or are produced. Pseudo-target names allow make to perform tasks not directly connected with the creation of a program, such as deleting old files or printing copies of source files. For example, the following dependency line removes old copies of the given object files when the pseudo-target name "cleanup" is given in the invocation of make.

```
cleanup:

        delete x.obj

        delete y.obj

        delete x.obj
```

Since no file exists for a given pseudo-target name, the target is always assumed to be out of date. Thus, the associated series of commands are always executed.

Make also has built-in pseudo-target names that modify its operation. The pseudo-target name ".IGNORE" causes make to continue after an error. This is the same as the -i option. (make also ignores errors for a given command if the command string begins with a hyphen (-).)

The pseudo-target name ".PRECIOUS" prevents dependents of the current target from being deleted when make is terminated by an error condition or user-interruption (Control C).

Using Macros

An important feature of a makefile is that it can contain macros. A macro is a short name that represents a file name or command option. The macros can be defined when you invoke make or in the makefile itself.

A macro definition is line containing an equal sign not preceded by a colon or a tab. The name (string of letters and digits) to the left of the equal sign (trailing blanks and tabs are stripped) is assigned the string of characters following the equal sign (leading blanks and tabs are stripped). The following are valid macro definitions:

```
CFLAGS = optimize(3) debug

LIBS =
```

The last definition assigns "LIBS" the null string. A macro that is never explicitly defined has the null string as its value. A macro is invoked by preceding the name by a dollar sign; macro names longer than one character must be placed in parentheses. The name of the macro is either the single character after the dollar sign or a name inside parentheses. The following are valid macro invocations:

```
$(CFLAGS)

$(xy)

$Z

$(Z)
```

The last two invocations are identical.

You may include a macro definition in a command line. A macro definition argument has the same form as a macro definition in a makefile. Macros in a command line override corresponding definitions found in the makefile. For example, the command:

```
make RELEASE=internal
```

assigns the option 'internal' to RELEASE.

Make has built-in macros that can be used when writing dependency lines. The following is a list of these macros:

$* Contains the name of the current target with the suffix removed. Thus, if the current target is test.obj, $*, contains test. It may be used in dependency lines that redefine the built-in rules.

$@ Contains the full path name of the current target. It may be used in dependency lines with user-defined target names.

$< Contains the file name of the dependent that is more recent than the given target.

Using the Built-In Rules

Make provides a set of built-in dependency lines, called built-in rules, that automatically check the targets and dependents given in a makefile and create up-to-date versions of these files if necessary. The built-in rules are identical to user-defined dependency lines except that they use the suffix of the file name as the target or dependent instead of the file name itself. For example, make automatically assumes that all files with the suffix .obj have dependent files with the suffix .C.

When no explicit dependency line is given in a makefile for a given file, make automatically checks the default dependents of the file, forming the name of the dependents by removing the suffix of the given file and appending the pre-defined dependent suffixes. If the given

file is out of date with respect to these default dependents, make searches for a built-in rule that defines how to create an up-to-date version of the file and executes it. For example, if the file x.obj is needed and there is an x.c in the description or directory, x.c is compiled.

The built-in rules are designed to reduce the size of your makefile. They provide the rules for creating common files from typical dependents. Reconsider the example given in "Creating a Makefile". In this example, the program 'test' depended on three object files, x.obj, y.obj, and z.obj. The files x.c and y.c also depended on the include file 'defs'. In the original example each dependency and corresponding command sequence was explicitly given. Many of these dependency lines were unnecessary, since the built-in rules could have been used instead. The following is all that is needed to show the relationships between these files:

```
test: x.obj y.obj z.obj

        BND286 x.obj, y.obj, z.obj, \

        /lib/ic286/clid2c.lib, \

        /lib/ic286/crmx2c.lib, \

        /lib/ic286/cnoflt2c.obj, \

        /lib/ic286/clib2c.lib, \

        /rmx286/lib/rmxifc.lib $(TYPE) nopack $(TYPE) nopack \

        segsize(stack(8192)) rc(dm(100,50000)) object($@)

x.obj y.obj: defs
```

In this makefile, test depends on three object files, and an explicit command is given showing how to update test. However, the second line merely shows that two object files depend on the include file defs. No explicitly command sequence is given on how to update these files if necessary. Instead, make uses the built-in rules to locate the desired c source files, compile these files, and create the necessary object files.